TURING 图灵程序设计丛书

Packt>

Generative Adversarial Networks Projects

生成对抗网络项目实战

[印] 凯拉什·阿伊瓦 著　倪琛 译

通过实际项目探索生成对抗网络构架，使用TensorFlow和Keras构建新一代生成模型


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

凯拉什 · 阿伊瓦

（Kailash Ahirwar）

机器学习解决方案平台Mate Labs的联合创始人兼首席技术官，与人合作发明了去中心化的分布式深度学习训练协议Raven Protocol，机器学习和深度学习爱好者，其研究工作涉及人工智能的许多领域，包括自然语言处理、计算机视觉，以及使用GAN进行生成建模。

倪琛

曾就读于法国贡比涅工程技术大学，现就读于伦敦大学金史密斯学院数据科学专业。

数字版权声明



图灵程序设计丛书

Generative Adversarial Networks Projects

生成对抗网络项目实战

[印] 凯拉什·阿伊瓦　著
倪　琛　译

人民邮电出版社
北　京

图书在版编目（ＣＩＰ）数据

生成对抗网络项目实战 / (印) 凯拉什·阿伊瓦
(Kailash Ahirwar) 著 ; 倪琛译. -- 北京 : 人民邮电
出版社, 2020.1
（图灵程序设计丛书）
ISBN 978-7-115-48544-1

Ⅰ. ①生… Ⅱ. ①凯… ②倪… Ⅲ. ①机器学习
Ⅳ. ①TP181

中国版本图书馆CIP数据核字(2019)第270343号

内 容 提 要

生成对抗网络（GAN）可以模拟任何数据分布方式，因而潜力巨大，为很多难以自动化的问题提供了解决途径。本书立足理论，着重实践，带领读者快速熟悉并上手 GAN。本书首先介绍构建高效项目所涉及的概念、工具和库，然后利用不同类型的数据集，依次构建 7 个 GAN 项目，训练并优化 GAN 模型。这些项目涵盖了各种流行方法，包括 3D-GAN、Age-cGAN、DCGAN、SRGAN、StackGAN、CycleGAN 和 pix2pix。

本书适合数据科学家、机器学习开发者、深度学习从业者及希望构建实际 GAN 模型的 AI 爱好者阅读。

◆ 著　　　　[印] 凯拉什·阿伊瓦
译　　　　倪　琛
责任编辑　杨　琳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：14
字数：337千字　　　　2020年 1 月第 1 版
印数：1－3 000册　　　　2020年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2019-4446号

定价：69.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

前　　言

生成对抗网络（generative adversarial network，GAN）可用于构建新一代模型，因为它可以模拟任何数据分布方式。它是目前发展最迅速的**机器学习**（machine learning，ML）领域之一，并且有很多相关的重要研发工作正在开展。本书将介绍神经网络模型无监督训练的相关技术，带领读者从零开始构建 7 个完整的 GAN 项目。

本书首先介绍构建高效项目所涉及的概念、工具和库，其后不同的项目会用到不同类型的数据集。每一章在复杂程度和操作难度上都逐步提升，最终帮助读者熟练掌握 GAN。

本书将介绍 3D-GAN、DCGAN、StackGAN、CycleGAN 等流行技术，并且通过实际实现来理解生成模型的架构和功能。

本书旨在介绍如何在工作或项目中构建、训练并优化完整的 GAN 模型。

读者对象

本书适合数据科学家、机器学习开发者、深度学习从业者，以及希望通过项目指南构建实际的 GAN 模型来检验自己的知识和专业技能的 AI 爱好者阅读。

本书内容

第 1 章，生成对抗网络简介。这一章首先介绍 GAN 相关概念，包括判别网络、生成网络、博弈论等。然后介绍生成网络和判别网络的架构与目标函数、GAN 的训练算法、Kullback-Leibler 散度和 Jensen-Shannon 散度、GAN 的评估矩阵、GAN 存在的各种问题、梯度消失和梯度爆炸问题、纳什均衡、批归一化，以及 GAN 正则化。

第 2 章，使用 3D-GAN 生成图形。这一章简单介绍 3D-GAN 和其架构细节。这一章会训练一个可以生成现实世界 3D 图形的 3D-GAN。编写代码获取 3D ShapeNets 数据集，进行数据清洗和训练预处理后，使用深度学习库 Keras 构建 3D-GAN 模型。

第 3 章，使用 cGAN 实现人脸老化。这一章介绍 cGAN（conditional generative adversarial

network，条件生成对抗网络）和 Age-cGAN。首先介绍数据准备过程，包括数据下载、数据清洗以及数据格式处理。届时会用到 IMDb Wiki Images 数据集。然后编写代码，使用 Keras 框架构建一个 Age-cGAN 模型，并在 IMDb Wiki Images 数据集上进行训练。最后，用训练好的模型生成图片，只需输入年龄作为参数，模型就可以生成一个人在不同年龄的面部图像。

第 4 章，使用 DCGAN 生成动画人物。这一章首先介绍 DCGAN 以及数据准备过程，包括获取动画人物的数据集、数据清洗以及训练预处理。我们会在 Jupyter Notebook 内使用 Keras 构建一个 DCGAN 模型。然后介绍训练 DCGAN 的各种技术，以及超参数调优。最后使用训练好的模型生成动画人物，并讨论 DCGAN 的实际应用。

第 5 章，使用 SRGAN 生成逼真图像。这一章介绍如何训练 SRGAN 生成逼真图像。训练流程的第一步是收集数据集，然后是数据清洗和数据格式处理。这一章会介绍如何收集数据集、清洗数据，以及将数据处理成训练所需的格式。

第 6 章，StackGAN：基于文本合成逼真图像。这一章首先介绍 StackGAN，然后介绍如何收集数据集、清理数据以及转换数据格式。数据准备好后，在 Jupyter Notebook 内编写代码用 Keras 构建 StackGAN，并在 CUB 数据集上训练该模型。训练好的模型可以基于文本生成逼真图像。最后讨论 StackGAN 在行业中的应用，以及在生产环境中的部署。

第 7 章，使用 CycleGAN 将绘画转换为照片。这一章介绍如何训练一个 CycleGAN 模型将绘画转换为照片。首先介绍 CycleGAN 及其各种用法，然后讲解数据收集、数据清洗和数据格式处理的各种技术，接着在 Jupyter Notebook 内使用 Keras 构建 CycleGAN，并在预备好的数据集上训练 CycleGAN 模型，之后检验模型将绘画转换为照片的水平，最后介绍 CycleGAN 的实际应用。

第 8 章，使用 cGAN 实现图像对图像变换。这一章介绍如何训练 cGAN 来实现图像对图像变换。首先介绍 cGAN 和各种数据处理技术，包括数据收集、数据清洗和数据格式处理，接着在 Jupyter Notebook 内使用 Keras 构建 cGAN，然后介绍如何在预备好的数据集上训练 cGAN。训练中会尝试不同的超参数。最后测试 cGAN，并讨论图像对图像变换的实际应用。

第 9 章，预测 GAN 的未来。介绍过 GAN 的基本原理并且完成了 7 个项目之后，最后这一章来预测 GAN 的前景：首先介绍近几年 GAN 应用所取得的成就和受欢迎程度，然后谈一下我对 GAN 未来的看法。

如何使用本书

阅读本书需要熟悉深度学习和 Keras，并对 TensorFlow 有一定了解。如果有 Python 3 的编程经验会更好。

下载示例代码文件

如果你是从 http://www.packtpub.com 网站购买的图书，登录自己的账号后就可以下载所有已购图书的示例代码。如果你是从其他地方购买的图书，请访问 http://www.packtpub.com/support 网站并注册，我们会将代码文件直接发送到你的电子邮箱。

你可以通过以下步骤下载代码文件。

(1) 在我们的网站上登录或注册。
(2) 选择 SUPPORT 标签。
(3) 点击 Code Downloads & Errata。
(4) 在 Search 框中输入书名并按屏幕上的提示操作。

文件下载后，使用以下工具的最新版本来解压缩或提取文件夹。

- WinRAR/7-Zip（Windows）
- Zipeg/iZip/UnRarX（Mac）
- 7-Zip/PeaZip（Linux）

本书代码也托管在 GitHub 上，访问 https://github.com/PacktPublishing/Generative-Adversarial-Networks-Projects 即可获取[①]。Packt 拥有丰富的图书和视频资源，相关代码见 GitHub 仓库：https://github.com/PacktPublishing/。欢迎查阅！

排版约定

本书使用了多种文本样式。

正文中的代码采用以下样式："使用 scipy 的 `loadmat()`函数来检索体素。"

代码块的样式如下所示。

```
import scipy.io as io
voxels = io.loadmat("path to .mat file")['instance']
```

命令行输入或输出如下所示。

```
pip install -r requirements.txt
```

黑体字：用于新术语或重要的词语。

① 可以直接访问本书中文版页面，下载本书项目的源代码：http://www.ituring.com.cn/book/2681。——编者注



此图标表示警告或需要特别注意的内容。



此图标表示提示或技巧。

联系我们

一般反馈：发送邮件至 feedback@packtpub.com 并在主题处注明书名。如果对于本书有任何疑问，请发送邮件至 questions@packtpub.com。

勘误：尽管我们尽力确保内容准确，但出错仍在所难免。如果你在书中发现错误，不管是文本还是代码，请告知我们，我们不胜感激。如果你发现任何错误，请访问 http://www.packtpub.com/submit-errata，选择书名，点击 Errata Submission Form 链接，并输入详细说明。

反盗版：如果你发现我们的作品在互联网上被以任何形式非法复制，请立即向我们提供地址或网站名称，非常感谢。请把可疑盗版材料的链接发至 copyright@packtpub.com。

成为作者：如果你掌握某个领域的专业知识，并且有兴趣写作图书，请访问 authors.packtpub.com。

评论

欢迎评论。阅读、使用本书后，请在购买网站上留下评论。这样潜在读者可以参考你的意见来决定是否购买，Packt 可以了解你对该产品的看法，作者也能看到你对本书的反馈。谢谢！

想了解关于 Packt 的更多信息，请访问 packtpub.com。

电子书

扫描如下二维码，即可购买本书电子版。



目　　录

第 1 章 生成对抗网络简介

1

本章介绍**生成对抗网络**（GAN），这是一种使用无监督机器学习算法生成数据的深度神经网络架构。2014 年，Ian Goodfellow、Yoshua Bengio 和 Aaron Courville 在论文“Generative Adversarial Nets”中首次提出了 GAN。GAN 有多种应用，包括图像生成和药物开发等。

本章会介绍 GAN 的核心组件及其工作原理、GAN 背后的重要概念和技术、GAN 的优势和缺陷，以及 GAN 的实际应用。

本章将探讨以下主题。

- 什么是 GAN
- GAN 架构
- GAN 相关的重要概念
- GAN 的不同变体
- GAN 的优缺点
- GAN 的实际应用

1.1 什么是 GAN

GAN 是一种由生成网络和判别网络组成的深度神经网络架构。通过在生成和判别之间多次循环，两个网络相互对抗，试图胜过对方，从而训练了彼此。

1.1.1 什么是生成网络

生成网络使用现有数据生成新数据，比如使用现有图像来生成新图像。生成网络的核心任务是从随机生成的由数字构成的向量（称为“潜在空间”，latent space）中生成数据（比如图像、视频、音频或文本）。在构建生成网络时需要明确该网络的目标，例如生成图像、文本、音频、视频，等等。

1.1.2　什么是判别网络

判别网络试图区分真实数据和由生成网络生成的数据。对于输入的数据，判别网络需要基于事先定义的类别对其分类。这可能是多分类或二分类。通常，GAN 中进行的是二分类。

1.1.3　GAN 通过对抗竞赛进行训练

GAN 中的网络通过对抗竞赛进行训练：两个网络互相竞争。例如用 GAN 生成艺术品赝品。

(1) 第一个网络，即生成网络，并未见过艺术品实物，但试图生成形似艺术品实物的作品。
(2) 第二个网络，即判别网络，试图判断一件艺术品是真品还是赝品。
(3) 生成网络在不断迭代中生成看起来更加真实的艺术品，试图骗过判别网络，让它相信这些生成的赝品是真品。
(4) 判别网络不断优化区分真假的标准，试图胜过生成网络。
(5) 在每轮迭代中，它们会将自己所做调整中的成功尝试反馈给对方，这就是 GAN 的训练过程。
(6) 最终，在判别网络的帮助下，生成网络已经训练得让判别网络无法区分哪件是真品、哪件是赝品了。

在该竞赛中，两个网络是同时受训的。当判别网络无法区分真品和赝品时，该网络就进入了一种名为“纳什均衡”的状态。本章稍后会详述。

1.2　GAN 的实际应用

GAN 的一些实际应用非常实用，例如下面这些。

- **图像生成**。在简单的图像数据上训练后的生成网络可生成逼真的图像。例如想生成新的狗狗图像，就可以在成千上万的狗狗图像的数据集上训练一个 GAN。训练完成之后，生成网络就可以生成一些不同于训练集的新图像。图像生成可用于市场营销、logo 制作、娱乐和社交媒体等领域。下一章会介绍动画人物面部图像的生成。
- **文本到图像的合成**。GAN 的一个有趣应用是基于文本生成图像。这对于电影行业来说很有用，因为 GAN 可以基于一段文本生成新的图像数据。对于漫画行业，它甚至可以自动生成一段故事。
- **人脸老化**。该应用对娱乐行业和监控领域都很有用，尤其是对于人脸验证方面，因为企业无须在员工的年龄增长之后更新安防系统了。Age-cGAN 可以生成不同年龄的图像，可用于构建强大的人脸验证模型。
- **图像到图像的变换**。图像到图像的变换可用于将白天拍摄的图像转换成夜晚拍摄的图像，将草图转换成绘画，将图像转换成毕加索或者梵高的风格，将航拍图自动转换成卫星图像，以及把马的图像转换成斑马的图像，等等。这些应用有助于节约时间，非常具有开创性。

- **视频合成**。GAN 也可以用于生成视频。用 GAN 生成内容快于人工创作，可以提高电影制作效率，也能帮助那些希望在业余时间制作创意视频的爱好者。
- **高清图像生成**。GAN 可以为用低像素相机拍摄的照片生成高清版，同时不损失任何关键细节。这有助于设计网站。
- **补全缺损图像**。如果图像缺失了某些部分，GAN 可以帮助恢复。

1.3　GAN 的具体架构

GAN 主要由两部分构成：生成网络和判别网络。每个网络都可以是任何神经网络，比如普通的**人工神经网络**（artificial neural network，ANN）、**卷积神经网络**（convolutional neural network，CNN）、**循环神经网络**（recurrent neural network，RNN）或者**长短期记忆**（long short term memory，LSTM）**网络**。判别网络则需要一些全连接层，并且以分类器收尾。

下面详细介绍 GAN 架构的组件。以一个简单的 GAN 为例。

1.3.1　生成网络的架构

这个简单的 GAN 中的生成网络是一个 5 层的简单前馈神经网络（feed-forward neural network），含 1 个输入层、3 个隐藏层以及 1 个输出层，具体配置见表 1-1。

表　1-1

层序号	层名称	配置
1	输入层	input_shape=(batch_size, 100), output_shape=(batch_size, 100)
2	全连接层	neurons=500, input_shape=(batch_size, 100), output_shape=(batch_size, 500)
3	全连接层	neurons=500, input_shape=(batch_size, 500), output_shape=(batch_size, 500)
4	全连接层	neurons=784, input_shape=(batch_size, 500), output_shape=(batch_size, 784)
5	输出层	input_shape=(batch_size, 784), output_shape=(batch_size, 28, 28)

上表列出了网络中输入层、隐藏层以及输出层的配置情况。

图 1-1 展示了生成网络里的张量（tensor）流，以及每一层输入和输出的张量的形状。

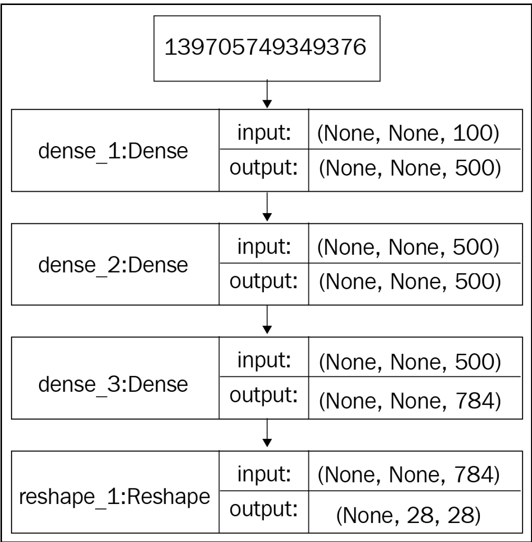


图 1-1　生成网络的架构

该前馈神经网络通过正向传播处理信息的过程如下。

- 输入层从正态分布采样一个 100 维的向量，不做任何修改，直接传递给第一个隐藏层。
- 3 个隐藏层分别是具有 500、500 和 784 个单元的全连接层。第 1 个隐藏层将一个形状为 `(batch_size, 100)`的张量变换成`(batch_size, 500)`。
- 第 2 个隐藏层（基于上一层输出的结果）将张量形状变换为`(batch_size, 500)`。
- 第 3 个隐藏层继续将张量形状变换为`(batch_size, 784)`。
- 最后的输出层将张量的形状从`(batch_size, 784)`变换为`(batch_size, 28, 28)`。这意味着该神经网络会生成一批图像，其中每张图像的形状为`(28, 28)`。

1.3.2　判别网络的架构

该 GAN 中的判别网络是一个 5 层的前馈神经网络，包括 1 个输入层、1 个输出层以及 3 个全连接层。判别网络是一个分类器，和生成网络有些区别。它会处理图像，然后输出该图像属于某个类别的概率。

图 1-2 展示了判别网络中的张量流，以及每一层输入和输出的张量的形状。

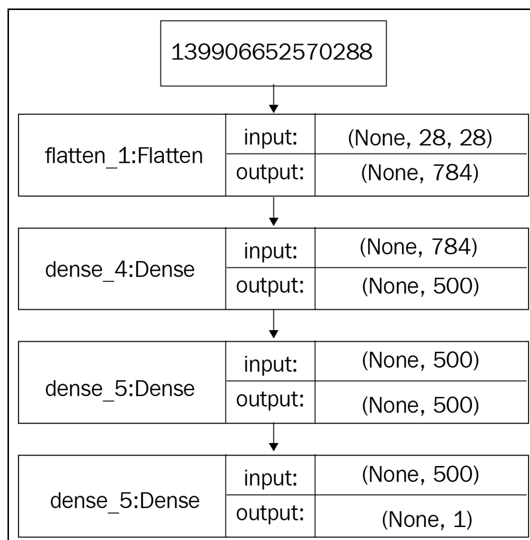


图 1-2　判别网络的架构

判别网络在训练过程中利用正向传播来处理数据的过程如下。

(1) 首先读取一个形状为 28×28 的张量输入。

(2) 输入层接收形状为`(batch_size, 28, 28)`的输入张量，不做任何修改，直接传递给第一个隐藏层（扁平化层）。

(3) 扁平化层将该张量转换成 784 维，然后将其传递给第一个隐藏全连接层。经过前两个隐藏层的处理，张量转换成了 500 维。

(4) 最后一层是输出层，也是全连接层，只有一个单元（神经元），使用 sigmoid 激活函数。它只输出 0 或 1：输出 0 意味着判别网络认为输入图像是假的；输出 1 意味着判别网络认为输入图像是真的。

1.3.3　GAN 相关重要概念

前面介绍了 GAN 的架构，下面简单介绍一些重要概念。首先介绍 KL（Kullback-Leibler）散度和 JS（Jensen-Shannon）散度，它们是评估模型质量的重要手段；然后介绍纳什均衡，这是在训练过程中希望达到的状态；最后重点介绍目标函数，理解目标函数有助于实现 GAN。

1. KL 散度

KL 散度，也称**相对熵**，用于判定两个概率分布之间的相似度。它可以测量一个概率分布 p 相对于另一个概率分布 q 的偏离。

如下公式用于计算两个概率分布 $p(x)$和 $q(x)$之间的 KL 散度。

$$D_{\mathrm{KL}}(p \parallel q)=\int_{x} p(x)\log\frac{p(x)}{q(x)}\mathrm{d}x$$

如果 $p(x)$和 $q(x)$处处相等，则此时 KL 散度为 0，达到最小值。

由于 KL 散度具有不对称性，因此不用于测量两个概率分布之间的距离，因此也不用作距离的度量（metric）。

2. JS 散度

JS 散度，也称**信息半径**（information radius，IRaD）或者**平均值总偏离**（total divergence to the average），是测量两个概率分布之间相似度的另一种方法。它基于 KL 散度，但具有对称性，可用于测量两个概率分布之间的距离。对 JS 散度开平方即可得到 JS 距离，所以它是一种距离度量。

计算两个概率分布 p 和 q 之间 JS 散度的公式如下。

$$D_{\mathrm{JS}}(p \parallel q)=\frac{1}{2}D_{\mathrm{KL}}(p \parallel \frac{p+q}{2})+\frac{1}{2}D_{\mathrm{KL}}(q \parallel \frac{p+q}{2})$$

其中，$(p+q)/2$ 是 p 和 q 的中点测度，D_{KL} 是 KL 散度。

介绍过了 KL 散度和 JS 散度，下面介绍 GAN 中的纳什均衡。

3. 纳什均衡

博弈论中的纳什均衡描述了一种在非合作博弈中可以达到的特殊状态。其中每个参与者都试图基于对其他参与者行为的预判，选择使自己获益最多的最佳策略。最终形成的局面是，所有参与者都基于其他参与者的选择，采取了对自己来说最佳的策略，此时已经无法通过改变策略获益了。这种状态就称为纳什均衡。

达到纳什均衡的一个著名例子是囚徒困境。两名犯罪嫌疑人（A 和 B）因为犯罪被逮捕了，分别关在不同的牢房，无法互相交流。检察官的证据只够将他们以较小的罪名定罪，而无法以主要罪行定罪（如果可以的话就会在监狱里关很久）。为了能以主要罪行定罪，检察官给他们提供了如下选择。

- 如果 A 和 B 同时检举对方的主要罪行，他们都会被关押 2 年。
- 如果 A 检举了 B 而 B 选择保持沉默，A 会被直接释放，而 B 会被关押 3 年（反之亦然）。
- 如果 A 和 B 都选择保持沉默，他们会因为较小罪名而都被关押 1 年。

从这 3 种结果来看，对于 A、B 两人整体而言，最佳结果显然是都选择保持沉默，然后都被关押 1 年。然而选择保持沉默的风险在于，他俩都不知道对方是否也会选择保持沉默。这样对于两人来说最佳策略其实是检举对方，因为在任何情况下这样做都是好处更大而惩罚更小。这样的局面一旦形成，任何罪犯都无法通过改变策略来获得任何好处，也就达到了纳什均衡。

4. 目标函数

为了使生成网络生成的图像能以假乱真，应尽量提高生成网络所生成数据和真实数据之间的相似度。可使用目标函数测量这种相似度。生成网络和判别网络各有目标函数，训练过程中也分别试图最小化各自的目标函数。GAN 最终的目标函数如下所示。

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

其中，$D(x)$是判别网络模型，$G(z)$是生成网络模型，$p(x)$是真实数据分布，$p(z)$是生成网络生成的数据分布，$\mathbb{E}$ 是期望输出。

在训练过程中，D（判别网络，discriminator）试图最大化公式的最终取值，而 G（生成网络，generator）试图最小化该值。如此训练出来的 GAN 中，生成网络和判别网络之间会达到一种平衡，此时模型即“收敛”了。这种平衡状态就是纳什均衡。训练完成之后，就得到了一个可以生成逼真图像的生成网络。

1.3.4 评分算法

GAN 的准确度容易计算。GAN 的目标函数不是均方误差（mean-square error）或者交叉熵（cross entropy）这样确定的函数，而是在训练过程中习得的。研究者们提出了多种可以测量模型准确度的评分算法，下面介绍其中几个。

1. Inception 分数

Inception 分数（IS）是应用最广泛的 GAN 评分算法。它使用一个在 Imagenet 上预训练过的 Inception V3 网络分别提取真实图像和生成图像的特征。Shane Barrat 和 Rishi Sharma 在论文“A Note on the Inception Score”中首次提出了该方法。IS 测量生成图片的质量和多样性。计算 IS 的公式如下。

$$\text{IS}(G) = \exp(\mathbb{E}_{\chi \sim p_g} D_{\text{KL}}(p(y \mid \chi) \parallel p(y)))$$

其中，p_g 表示一个概率分布，$\chi \sim p_g$ 表示 χ 是该概率分布中的一个抽样。$p(y \mid \chi)$是条件类别分布，$p(y)$是边缘类别分布。

计算 Inception 分数的步骤如下。

(1) 首先从模型生成的图像中抽取 N 个样本，记为(χ^i)。

(2) 然后使用如下公式构建边缘类别分布。

$$p(y) = \int_{\chi} p(y \mid \chi) p_g(\chi)$$

(3) 接着使用如下公式计算 KL 散度以及期望值。

$$\text{IS}(G) = \exp(\mathbb{E}_{\chi \sim p_g} D_{\text{KL}}(p(y \mid \chi) \parallel p(y)))$$

(4) 最后计算上述结果的指数，即可得到 IS。

IS 越高，说明模型质量越好。IS 虽然是重要的测度（measure），却也存在一些问题。比如模型对于每个类别只生成一张图像，其 IS 仍然可以很高，但这样的模型缺乏多样性。为了解决该问题，人们又提出了其他一些性能测度，稍后会介绍其中一种。

2. Fréchet Inception 距离

为了克服 Inception 分数的一些缺陷，Martin Heusel 等人在论文“GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”中提出了 Fréchet Inception 距离（Fréchet Inception Distance，FID）。

计算 FID 分数的公式如下。

$$\text{FID} = \left\| \mu_r - \mu_g \right\|^2 + T_r(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2})$$

上面的公式表示真实图像集 x 和生成图像集 g 之间的 FID 分数。要计算 FID 分数，首先抽取 Inception 网络的一个中间层的特征映射，构建一个多元正态分布来学习这些特征映射的概率分布，然后使用该多元正态分布的均值 μ 和协方差 Σ 来计算 FID 分数。FID 分数越低，说明模型的质量越好，其生成多样化的、高质量的图像的能力就越强。完美的生成模型的 FID 分数应该为 0。FID 分数优于 IS 之处在于对噪声的抵抗力较好，并且可以更好地测量图像的多样性。



关于 FID 的 TensorFlow 实现，可访问 https://www.tensorflow.org/api_docs/python/tf/contrib/gan/eval/frechet_classifier_distance。学术界和业界的研究者还提出了其他评分算法，本书没有涉及。继续下面的内容之前，请先了解评分算法 Mode 分数。

1.4　GAN 变体

目前，GAN 的变体数以千计，并且这个数字还在快速增长。下面简单介绍 6 种流行的 GAN 架构，后续章节会详述。

1.4.1　深度卷积生成对抗网络

Alec Radford、Luke Metz 和 Soumith Chintala 在论文“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”中首次提出了深度卷积生成对抗网络（deep convolutional GAN，DCGAN）。普通 GAN 通常不包含 CNN，DCGAN 将其引入了 GAN。第 3 章会介绍如何使用 DCGAN 生成动画人物面部图像。

1.4.2 StackGAN

StackGAN 是由 Han Zhang、Tao Xu、Hongsheng Li 等人在论文“StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks”中首次提出的。他们使用 StackGAN 进行文本到图像的合成，效果极佳。StackGAN 可以基于文本生成逼真图像。第 6 章会介绍如何使用 StackGAN 基于文本生成逼真的图像。

1.4.3 CycleGAN

CycleGAN 是由 Jun-Yan Zhu、Taesung Park、Phillip Isola 和 Alexei A. Efros 在论文“Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”中首次提出的。CycleGAN 有一些非常有趣的应用，例如将照片转换成绘画（或者反过来），将夏天拍摄的照片转换成冬天拍摄的照片（或者反过来），以及将马的图像转换为斑马的图像（或者反过来）。第 7 章会介绍如何使用 CycleGAN 将绘画转换成照片。

1.4.4 3D-GAN

3D-GAN 是由 Jiajun Wu、Chengkai Zhang、Tianfan Xue、William T. Freeman 以及 Joshua B. Tenenbaum 在论文“Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling”中首次提出的。生成物体的 3D 模型在制造业和 3D 模型产业都有广泛用途。在物体的 3D 模型数据上训练之后，3D-GAN 可以为不同物体生成新的 3D 模型。第 2 章会介绍如何使用 3D-GAN 生成物体的 3D 模型。

1.4.5 Age-cGAN

Age-cGAN 是由 Grigory Antipov、Moez Baccouche 和 Jean-Luc Dugelay 在论文“Face Aging with Conditional Generative Adversarial Networks”中首次提出的。人脸老化用途广泛，包括跨年龄人脸识别、找寻失踪儿童以及娱乐应用等。第 3 章会介绍如何训练可以生成特定年龄的人脸图像的 cGAN。

1.4.6 pix2pix

pix2pix 是由 Phillip Isola、Jun-Yan Zhu、Tinghui Zhou 和 Alexei A. Efros 在论文“Image-to-Image Translation with Conditional Adversarial Networks”中首次提出的。pix2pix 网络和 CycleGAN 有相似的应用，可以将建筑标签转换成建筑图像（第 8 章有相关示例），将黑白图像转换成彩色图像，将白天拍摄的照片转换成夜间拍摄的照片，将草图转换成照片，以及将航拍图像转换成地图图像。



如感兴趣，可阅读 Avinash Hindupur 的文章“GAN Zoo”，里面列举了现有的所有 GAN。

1.5　GAN 的优势

相比其他监督学习方法或者无监督学习方法，GAN 的优势如下。

- **GAN 是无监督学习方法**。带标注数据需要人工制作，非常耗时。GAN 不需要带标注数据，而可以通过无标注数据进行训练，学习数据的内在表现形式。
- **GAN 可以生成数据**。GAN 可以生成能跟真实数据媲美的数据，应用潜力巨大。GAN 可以生成图像、文本、音频和视频等，并且和真实数据相差无几。用 GAN 生成图像可应用于市场营销、电子商务、游戏、广告等很多行业。
- **GAN 可以学习数据的概率密度分布**。GAN 可以学习数据的内在表现形式。前面提到了 GAN 可以学习混乱而复杂的数据概率分布，有助于解决机器学习领域的很多问题。
- **训练后的判别网络是分类器**。GAN 训练完成之后会得到一个判别网络和一个生成网络，而判别网络可用作分类器。

1.6　训练 GAN 的问题

像很多技术那样，GAN 也存在一些问题。这些问题通常与训练过程有关，包括模式塌陷、内部协变量转移以及梯度消失等。下面具体探讨这些问题。

1.6.1　模式塌陷

模式塌陷问题指的是生成网络所生成的样本之间差异不大，有时甚至始终只生成同样的图像。有一些概率分布是多峰的（multimodal），构造十分复杂。数据可能是通过不同类型的观测得来的，因此样本中可能会暗含一些细类，每个细类下的样本之间比较相似。这样会导致数据的概率分布出现多个“峰”，每个峰对应一个细类。如果数据的概率分布是多峰的，GAN 有时就会出现模式塌陷问题，无法成功构建模型。如果生成的所有样本几乎都相同，这种情况就被称为“完全塌陷”。

解决模式塌陷问题有多种方法，例如：

- 针对不同的峰训练不同的 GAN 模型；
- 使用多样化的数据训练 GAN。

1.6.2　梯度消失

在反向传播过程中，梯度从最后一层反向流动到第一层，并且会越来越小。有时梯度过小会导致前几层的学习速度非常慢，或者根本无法学习。在这种情况下，梯度无法改变前几层的权重值，所以网络前几层的训练没有任何效果。该问题称作**梯度消失**。

如果使用基于梯度的优化方法训练更大的神经网络，问题会更严重。基于梯度的优化方法是通过计算参数值上的小幅变动对神经网络输出的影响来优化参数的。如果参数值的变动对神经网络的输出影响非常小，优化算法对参数值的调整也会很小，所以神经网络的学习就停滞了。

使用像 sigmoid 和 tanh 这样的激活函数，梯度消失问题同样会存在。sigmoid 激活函数将输出值限定在 0 到 1，较大的输入会映射为接近 1 的数值，较小的输入或者负数输入会映射为接近 0 的数值。类似地，tanh 激活函数将输出值限定在–1 到 1，较大的输入会映射为接近 1 的数值，较小的数值会映射为接近–1 的数值。反向传播过程需要用到微分上的链式法则，会产生乘数效应。当反向传播到达神经网络的前几层的时候，梯度已经非常小了，就会出现梯度消失的问题。

为了解决该问题，可以使用 ReLU、LeakyReLU 或者 PReLU 等激活函数。这些激活函数的梯度不会在反向传播过程中被耗散掉，因此可以有效训练神经网络。另一个解决办法是使用批归一化，该方法对网络中隐藏层接收的输入先进行归一化，然后才传递给隐藏层。

1.6.3 内部协变量转移

内部协变量转移问题之所以产生，是因为神经网络输入数据的概率分布发生了变化。输入数据的概率分布改变之后，隐藏层会试图适应新的概率分布，训练速度因此放缓，需要很长时间才会收敛到全局最小值。神经网络输入数据的概率分布和该网络之前接触的数据概率分布之间差异过大是问题根源。解决方法包括批归一化以及其他归一化技术，下面会探讨这些技术。

1.7 解决 GAN 训练稳定性问题

GAN 可能出现训练不稳定的问题，严重时会导致 GAN 永远无法在某些数据集上收敛。下面介绍可以提高 GAN 稳定性的一些办法。

1.7.1 特征匹配

在 GAN 的训练过程中，判别网络的目标函数需要最大化，而生成网络的目标函数需要最小化。这样的目标函数存在一些严重的缺陷，比如没有考虑生成数据和真实数据的分布特征。

特征匹配是 Tim Salimans 和 Ian Goodfellow 等人在论文“Improved Techniques for Training GANs”中首次提出的技术，引入了一种新的目标函数来提高 GAN 的收敛能力。使用新的目标函数，便于生成网络生成在分布特征上和真实数据更为接近的数据。

在特征映射技术中，判别网络不再输出二元标签，而改为输出某个中间层对于输入数据的“激活映射”，也称“特征映射”。这样可以训练判别网络学习真实数据的分布特征，并且使用这些特征来区分真实数据和虚假数据。

首先引入一些计法，以便从数学上讲解该方法。

- $f(x)$：判别网络某中间层对于真实数据的激活映射或者特征映射。
- $f(G(z))$：判别网络某中间层对于生成网络生成的数据的激活映射或者特征映射。

新的目标函数如下所示。

$$\left\|\mathbb{E}_{x\sim p_{\text{data}}} f(x) - \mathbb{E}_{z\sim p_z(z)} f(G(z))\right\|_2^2$$

使用该目标函数可以获得更好的结果，但不能保证收敛。

1.7.2　小批量判别

小批量判别也能让 GAN 训练更稳定。该方法由 Ian Goodfellow 等人在论文“Improved Techniques for Training GANs”中首次提出。介绍该方法之前，先说明该方法所解决的问题。训练 GAN 的过程中，如果判别网络接收的输入图像彼此不相关，它们的梯度之间无法产生联系，判别网络就无法学习如何区分生成网络生成的不同种类的图像。这会导致前面介绍过的模式塌陷问题[①]。小批量判别可以解决该问题。图 1-3 很好地展现了该过程。

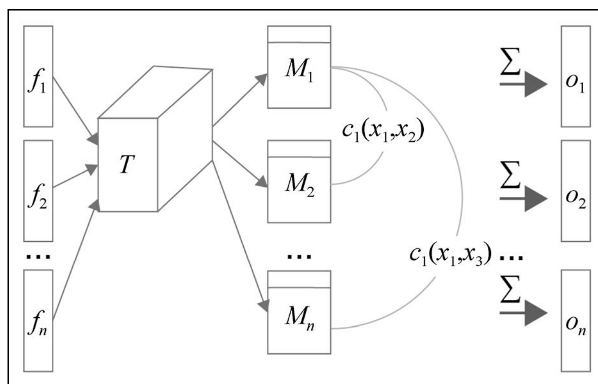


图 1-3　通过小批量判别解决模式塌陷

小批量判别是一个多步骤过程。为神经网络添加小批量判别的步骤如下。

(1) 抽取样本的特征映射，然后乘以张量 $T \in R^{A\times B\times C}$，得到矩阵 $\boldsymbol{M}_i \in R^{A\times B}$。

(2) 使用如下公式计算 $\boldsymbol{M}_i$ 矩阵各行之间的 L1 距离。

$$c_b(x_i, x_j) = \exp(-\left\|M_{i,b} - M_{j,b}\right\|_{L1}) \in \mathbb{R}$$

① 因为生成网络得不到判别网络的相应反馈，也就不知道如何生成多样化的图像。——译者注

(3) 对于某个输入数据 x_i，计算步骤(2)得到的所有距离之和。

$$o(x_i)_b = \sum_{j=1}^{n} c_b(x_i, x_j) \in \mathbb{R}$$

(4) 将 $o(x_i)$和 $f(x_i)$拼接起来，作为输入传递给神经网络的下一层。

$$o(x_i) = [o(x_i)_1, o(x_i)_2, \cdots, o(x_i)_B] \in \mathbb{R}^B$$
$$o(X) \in \mathbb{R}^{n \times B}$$

为了从数学上理解，下面具体解释这些表达式。

- ❑ $f(x_i)$：判别网络某个中间层对于第 i 个样本的激活映射或者特征映射。
- ❑ $T \in R^{A \times B \times C}$：一个三维张量，用于和 $f(x_i)$相乘。
- ❑ $\boldsymbol{M}_i \in R^{A \times B}$：张量 T 和 $f(x_i)$相乘生成的矩阵。
- ❑ $o(x_i)$：对于某样本 x_i，计算其 $\boldsymbol{M}_i$所有行之间 L1 距离之和。

小批量判别有助于避免峰坍塌，提高训练的稳定性。

1.7.3 历史平均

历史平均计算历史参数的平均值，然后将该值分别加入生成网络和判别网络的成本函数中。该方法是由 Ian Goodfellow 等人在论文“Improved Techniques for Training GANs”中首次提出的。

计算历史平均的公式如下。

$$\left\| \theta - \frac{1}{t} \sum_{i=1}^{t} \theta[i] \right\|^2$$

公式中的 $\theta[i]$ 是全部参数在某一时刻 i 的取值。该方法也可以提高 GAN 的稳定性。

1.7.4 单面标签平滑

在之前的例子中，分类器的标签（即目标值）只可取 0 或者 1，0 代表假图像，1 代表真图像。在这样的设定下，GAN 很容易受到对抗样本问题的影响。对抗样本是一种特殊的输入数据，如果在输入数据中叠加对抗样本，原本可以正常进行分类的神经网络会产生错误的分类结果。标签平滑技术可以为判别网络提供平滑后的标签，比如使用 0.9（真）、0.8（真）、0.1（假）或者 0.2（假），而不是对所有样本都标注 1（真）或者 0（假）。真图像和假图像的目标值（标签值）都需要进行平滑处理。标签平滑有助于降低 GAN 出现对抗样本的风险。具体做法是为图像标注 0.9、0.8、0.7 以及 0.1、0.2、0.3 等标签。关于标签平滑的更多信息，可参考论文“Improved Techniques for Training GANs”。

1.7.5　批归一化

批归一化技术是将特征向量归一化，使其均值为 0，方差为 1。该技术可提高学习过程的稳定性，以及解决权重值初始化效果差的问题。将该技术作为预处理步骤应用于神经网络的隐藏层，有助于缓解内部协变量转移的问题。

批归一化是在 2015 年由 Ioffe 和 Szegedy 在论文“Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”中首次提出的。

批归一化的优势如下。

- **缓解内部协变量转移问题**。批归一化可以通过数值归一化缓解内部协变量转移问题。
- **加速训练**。如果神经网络接收的输入值呈正态分布，那么可提高该网络的训练速度。批归一化可以提升神经网络内部各层所接收数值的质量，提高训练的整体速度（不过由于添加了额外的计算，每轮迭代会变慢）。
- **提高准确性**。批归一化可以提高模型的准确性。
- **提高学习速率**。神经网络通常需要使用较低的学习速率训练，需要很长时间才会收敛。借助批归一化，可以采用更高的学习速率，使神经网络更快地达到全局最小值。
- **减少对随机失活技术的依赖**。使用随机失活会损失神经网络内部各层的一些关键信息。批归一化可以起到正则项的作用，避免使用随机失活层。

批归一化需要对所有隐藏层应用，而不仅仅是输入层使用。

1.7.6　实例归一化

前面提过，批归一化基于一批数据的整体信息进行归一化。实例归一化有所不同，对一个特征映射进行归一化时只使用该特征映射的信息。实例归一化是由 Dmitry Ulyanov 和 Andrea Vedaldi 在论文“Instance Normalization: The Missing Ingredient for Fast Stylization”中首次提出的。

1.8　小结

本章介绍了 GAN、GAN 标准架构的组件以及 GAN 变体。基于 GAN 的基本概念，本章还介绍了有关 GAN 构建和功能的深层概念、GAN 的优势和缺陷，以及克服缺陷的一些办法，最后介绍了 GAN 的各种实际应用。

基于本章讲述的 GAN 关键知识，下一章会介绍如何使用 GAN 生成不同的图形。

第 2 章

使用 3D-GAN 生成图形

3D-GAN 是一种可生成 3D 图形的 GAN 架构。生成 3D 图形特别复杂，因为 3D 图像处理涉及许多难点。3D-GAN 能生成不同类型的逼真 3D 图形，由 Jiajun Wu、Chengkai Zhang 和 Tianfan Xue 等人在论文“Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling”中首次提出。本章将使用 Keras 框架实现 3D-GAN。

本章将讨论以下主题。

- 3D-GAN 简介
- 创建项目
- 准备数据
- 3D-GAN 的 Keras 实现
- 训练 3D-GAN
- 超参数优化
- 3D-GAN 的实际应用

2.1 3D-GAN 简介

类似于 StackGAN、CycleGAN 以及超分辨率生成对抗网络（super-resolution generative adversarial network，SRGAN），3D-GAN 也是 GAN 的一个变体。和普通 GAN 相同，它具有一个生成网络和一个判别网络。这两个网络都使用 3D 卷积层而不是 2D 卷积层。如果提供的数据量足够大，它可以学会生成视觉效果很好的 3D 图形。

深入探讨 3D-GAN 之前，先简单介绍一下 3D 卷积。

2.1.1 3D 卷积

简单说来，3D 卷积运算是对输入数据在 x、y 和 z 三个维度上应用 3D 过滤器。该运算会创建一个 3D 特征映射的堆叠列表。输出的形状类似于一个立方体或长方体。图 2-1 演示了一个 3D

卷积运算。左边立方体中标亮的部分是输入数据，中间是形状为(3, 3, 3)的卷积核，右边是卷积运算的输出。

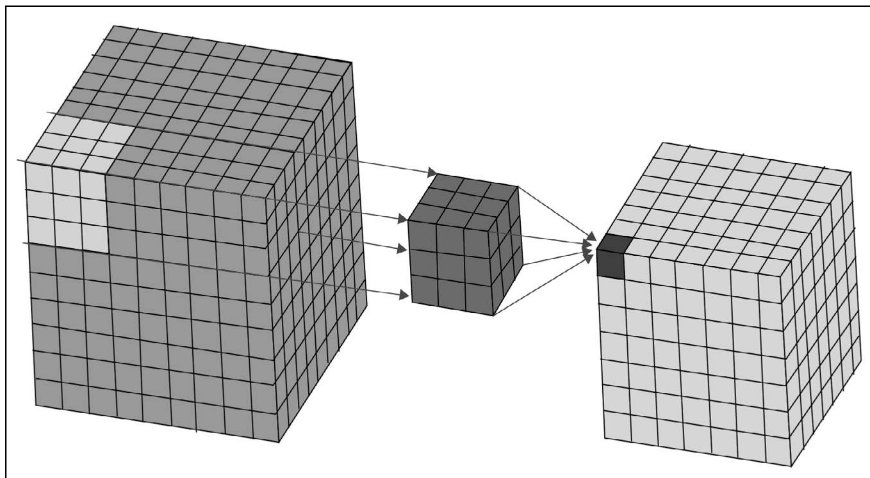


图 2-1　一个 3D 卷积运算

介绍过了 3D 卷积的基础知识，下面介绍 3D-GAN 架构。

2.1.2　3D-GAN 架构

3D-GAN 中的两个神经网络都是深度卷积神经网络。生成网络仍是上采样网络，它对噪声向量（概率潜在空间中的一个向量）进行上采样，生成一个和输入图像在长、宽、高和通道上形状相似的 3D 图像。判别网络是下采样网络，通过一系列 3D 卷积运算和一个全连接层来判断输入数据的真假。

下面介绍生成网络和判别网络的架构。

1. 生成网络的架构

生成网络包含 5 个体积型完全卷积层，配置如下。

- 卷积层数量：5
- 过滤器数量：512，256，128，64，1
- 卷积核大小：4×4×4，4×4×4，4×4×4，4×4×4，4×4×4
- 步长：1，2，2，2，2 或(1, 1)，(2, 2)，(2, 2)，(2, 2)，(2, 2)
- 是否使用批归一化：是，是，是，是，否
- 激活函数：ReLU，ReLU，ReLU，ReLU，sigmoid
- 是否使用池化层：否，否，否，否，否

- **是否是线性层**：否，否，否，否，否

生成网络的输入和输出如下。

- **输入**：从概率潜在空间采样的 200 维向量。
- **输出**：形状为 64×64×64 的 3D 图像。

图 2-2 展示的是生成网络的架构。

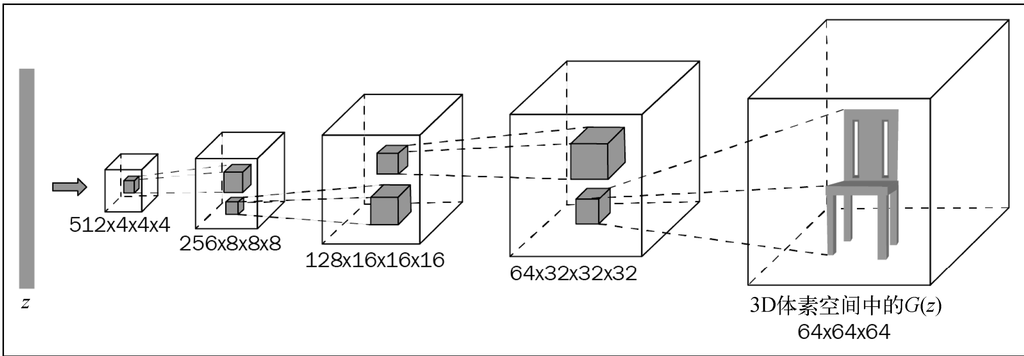


图 2-2 生成网络的架构

图 2-3 展示了生成网络中各层的张量流，以及输入和输出张量的形状，以便于理解。

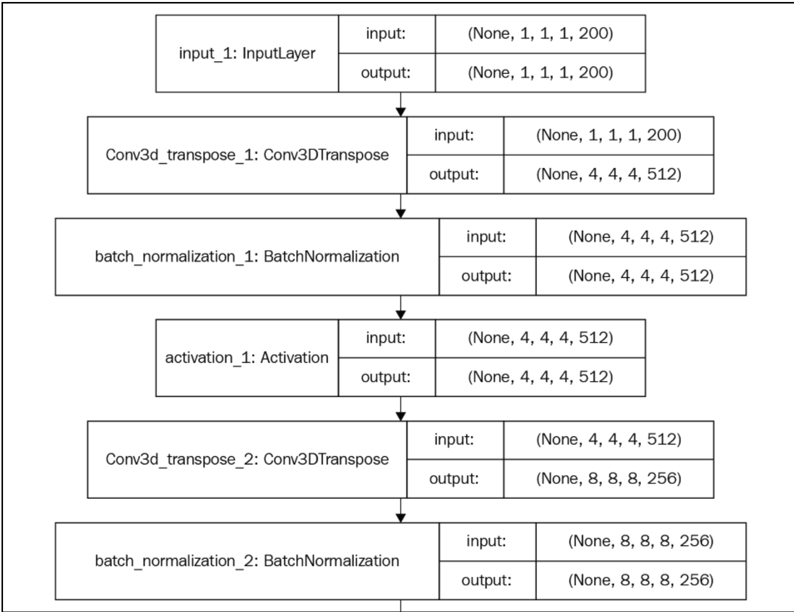


图 2-3 生成网络中各层的张量流，以及输入和输出张量的形状

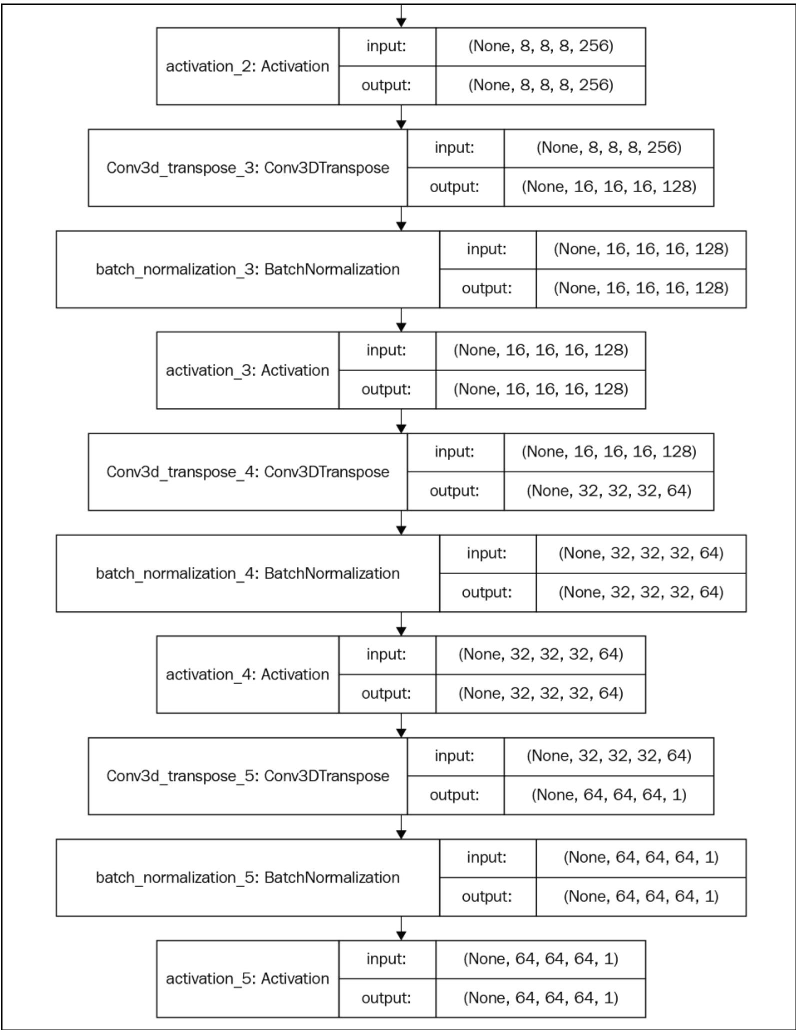


图 2-3 （续）



完全卷积网络不以全连接层收尾，它完全是由卷积层组成的，但像以全连接层收尾的卷积网络那样支持端到端的训练。生成网络中没有池化层。

2. 判别网络的架构

判别网络包含 5 个体积型卷积层，配置如下。

- 3D 卷积层数量：5
- 过滤器数量：64，128，256，512，1
- 卷积核大小：4，4，4，4，4

❑ **步长**：2，2，2，2，1
❑ **激活函数**：LeakyReLU，LeakyReLU，LeakyReLU，LeakyReLU，sigmoid
❑ **是否使用批归一化**：是，是，是，是，否
❑ **是否使用池化层**：否，否，否，否，否
❑ **是否是线性层**：否，否，否，否，否

判别网络的输入和输出如下。

❑ **输入**：形状为`(64, 64, 64)`的 3D 图像。
❑ **输出**：输入图像属于“真实”类别或“虚假”类别的概率。

图 2-4 展示了判别网络中各层的张量流，以及输入和输出张量的形状，以便于理解。

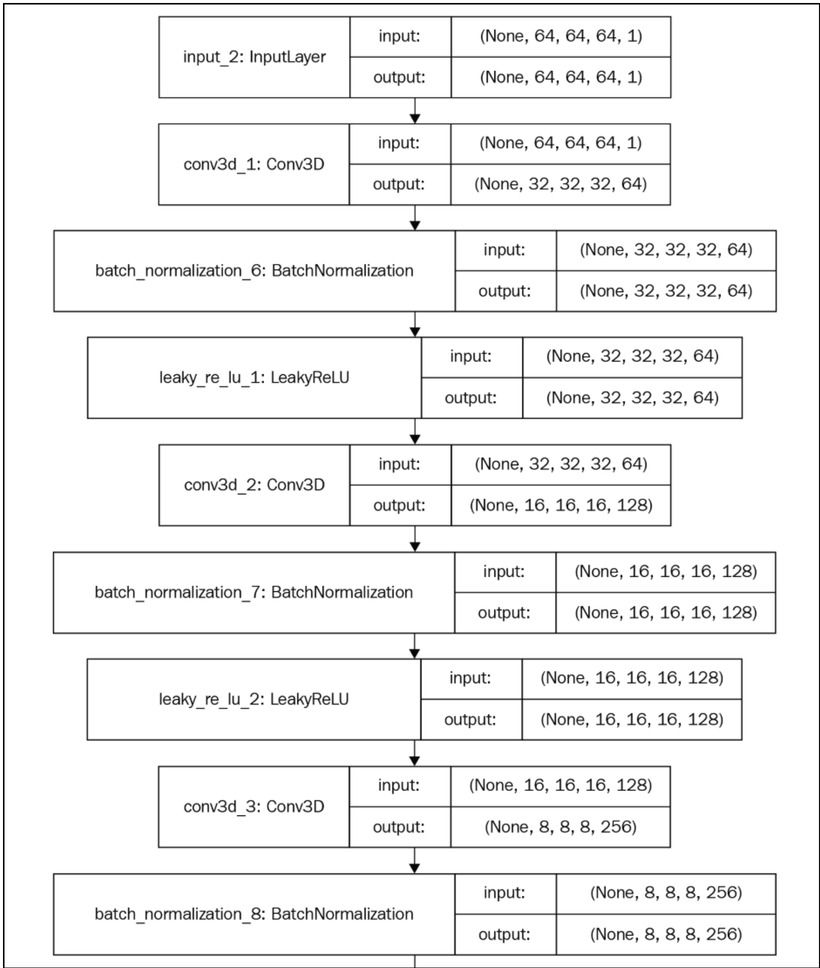


图 2-4　判别网络中各层的张量流，以及输入和输出张量的形状

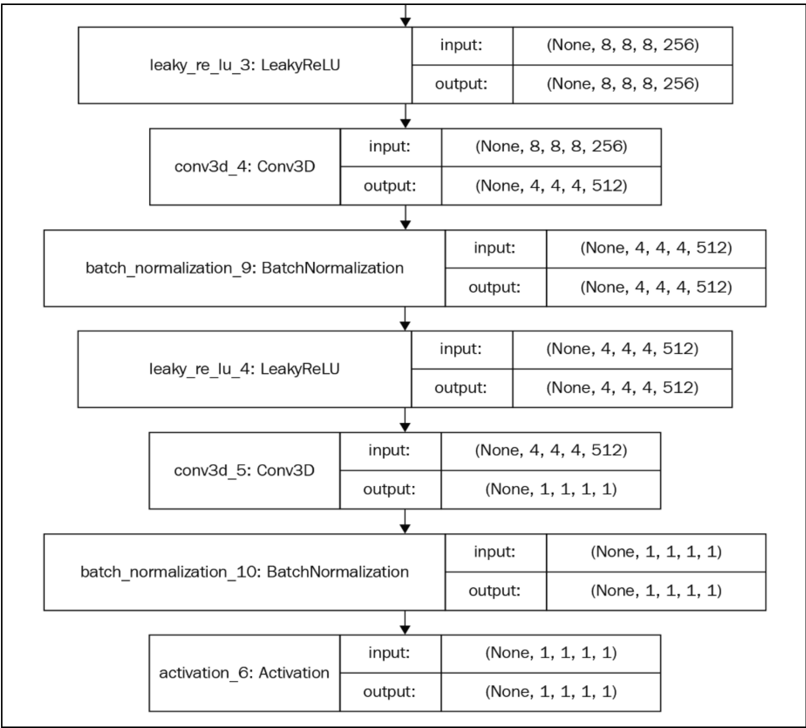


图 2-4　（续）

判别网络相当于生成网络的镜像。一个重要的区别是，判别网络使用的激活函数是 LeakyReLU 而不是 ReLU，而且判别网络最后的 sigmoid 层用于进行二分类，估测输入图像的真假。最后一层没有批归一化层，但其他层都使用批归一化对输入进行正则化。

2.1.3　目标函数

目标函数是训练 3D-GAN 的主要手段。它提供了用于计算梯度并更新权重值的损失值。3D-GAN 的对抗损失函数如下。

$$L_{\text{3D-GAN}} = \log D(x) + \log(1 - D(G(z)))$$

其中 $\log(D(x))$是二元交叉熵损失（也称“分类损失”），$\log(1-D(G(z)))$是对抗损失，z 是概率空间 $p(z)$的潜在向量，$D(x)$是判别网络的输出，$G(z)$是生成网络的输出。

2.1.4　训练 3D-GAN

训练 3D-GAN 和训练普通 GAN 类似。3D-GAN 的训练步骤如下。

(1) 从高斯（正态）分布中采样一个 200 维的噪声向量。

(2) 使用生成网络生成一张假图像。
(3) 使用（从真实数据中采样的）真实图像和生成网络生成的假图像训练判别网络。
(4) 使用对抗模型训练生成网络，不训练判别网络。
(5) 按指定轮数重复上述步骤。

稍后会详细介绍这些步骤。下面先创建一个项目。

2.2 创建项目

本项目的源代码可从 GitHub 上获取，地址如下：https://github.com/PacktPublishing/Generative-Adversarial-Networks-Projects。

运行如下命令创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 然后从当前目录切换到 Chapter02 目录。

```
cd Chapter02
```

(3) 接着为本项目创建一个 Python 虚拟环境。

```
virtualenv venv
```

(4) 完成之后，启用该虚拟环境。

```
source venv/bin/activate
```

(5) 最后，安装 requirements.txt 文件中列出的全部所需程序。

```
pip install -r requirements.txt
```

这样就创建好了项目。如需了解更多信息，请参考代码目录中的 README.md 文件。

2.3 准备数据

本章使用 3D ShapeNets 数据集，下载地址为：http://3dshapenets.cs.princeton.edu/3DShapeNetsCode.zip。该数据集由 Wu 和 Song 等人发布，包含经过得当标注的 40 个类别的 3D 物体形状。本项目会使用文件夹中的体积型数据，稍后详细介绍。下面下载、提取并探索该数据集。



3D ShapeNets 数据集仅用于学术研究。如想商用，需征求论文作者同意，邮箱地址为：shurans@cs.princeton.edu。

2.3.1　下载并提取数据集

运行以下命令，下载并提取数据集。

(1) 首先通过以下地址下载 3D ShapeNets 数据集。

```
wget http://3dshapenets.cs.princeton.edu/3DShapeNetsCode.zip
```

(2) 下载完成之后，运行如下命令将文件提取至恰当的目录。

```
unzip 3DShapeNetsCode.zip
```

数据集的下载和提取就完成了。数据集中包含.mat（MATLAB）格式的图像。所有图像都是 3D 图像。下面介绍体素，即 3D 空间中的点。

2.3.2　探索数据集

为了更好地理解数据集，需要将其中的 3D 图像可视化。下面介绍什么是体素，然后加载一张 3D 图像并将其可视化。

1. 什么是体素

体素即体积像素（volume pixel），是三维空间中的点。体素通过 x、y、z 这 3 个方向的坐标定义了位置。体素是 3D 图像表示的基本单元，主要应用于 CAT 扫描、X 光以及 MRI，可以创建人体以及其他物体的精确 3D 模型。掌握体素这个概念有助于处理 3D 图像，因为它是 3D 图像的组成材料。图 2-5 直观展示了 3D 图像中的体素。

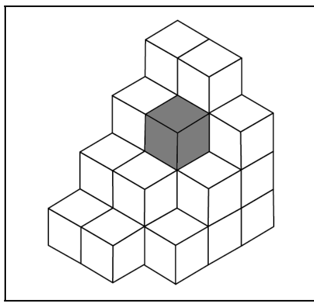


图 2-5　一张 3D 图像中的一系列体素，阴影区域表示的是一个体素

上图展现了堆叠的体素，其中的灰色立方体表示一个体素。介绍过了体素，下面加载 3D 图像并将其可视化。

2. 加载并可视化 3D 图像

3D ShapeNets 数据集包含了各种物体的 CAD 模型，文件格式为.mat。下面将这些.mat 文件

转换为 NumPy ndarray 形式，并把一张 3D 图像可视化，直观展现该数据集。

执行如下代码，从.mat 文件中加载一张 3D 图像。

(1) 使用 scipy 库中的 `loadmat()` 函数提取 `voxels` 变量。代码如下：

```
import scipy.io as io
voxels = io.loadmat("path to .mat file")['instance']
```


(2) 所加载的 3D 图像形状为 30×30×30，而所使用的网络要求图像的形状为 64×64×64。使用 NumPy 的 `pad()` 方法将 3D 图像的大小增加到 32×32×32。`pad()` 方法接收 4 个参数，分别为实际体素的 ndarray、每个轴两边需要填充值的数量、模式值（使用常数）以及需要填充的常数值。

```
import numpy as np
voxels = np.pad(voxels, (1, 1), 'constant', constant_values=(0, 0))
```

(3) 接着使用 scipy.ndimage 模块中的 `zoom()` 函数将该 3D 图像形状转换成 64×64×64。

```
import scipy.ndimage as nd
voxels = nd.zoom(voxels, (2, 2, 2), mode='constant', order=0)
```

该项目使用的神经网络要求图像的形状是 64×64×64，这就是需要将 3D 图像转换成该形状的原因。

3. 将 3D 图像可视化

下面使用 matplotlib 对 3D 图像进行可视化，代码如下。

(1) 首先创建一张 matplotlib 图，并为其添加一张子图。

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_aspect('equal')
```

(2) 然后向图中添加 `voxels` 变量。

```
ax.voxels(voxels, edgecolor="red")
```

(3) 生成图像并保存为图片，以便稍后查看。

```
plt.show()
plt.savefig(file_path)
```

图 2-6 展示了 3D 平面中的一架飞机。

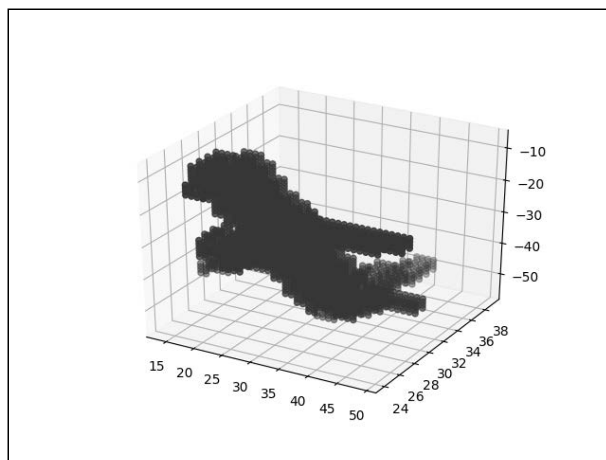


图 2-6　3D 平面中的一架飞机

图 2-7 展示了 3D 平面中的一张桌子。

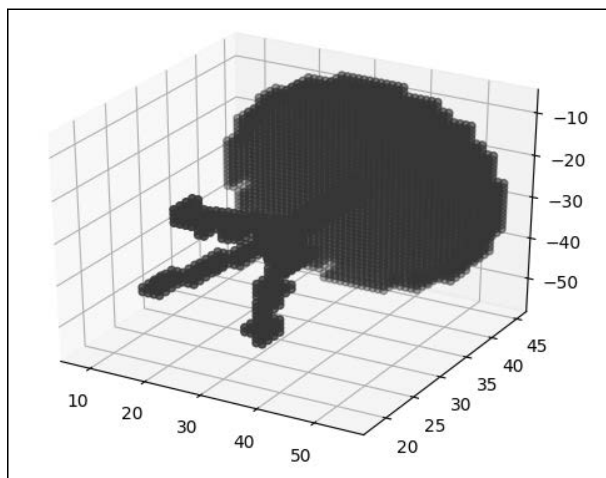


图 2-7　3D 平面中的一张桌子

图 2-8 展示了 3D 平面中的一把椅子。

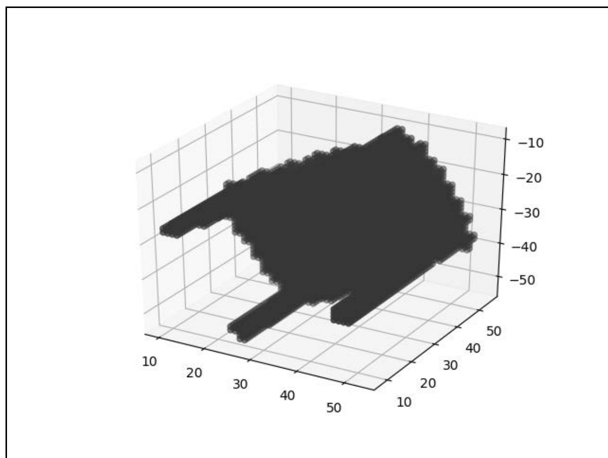


图 2-8　3D 平面中的一把椅子

至此，下载、提取并探索了数据集，还介绍了体素的使用方式。下面使用 Keras 框架实现一个 3D-GAN。

2.4　3D-GAN 的 Keras 实现

下面使用 Keras 框架实现生成网络和判别网络。需要创建两个 Keras 模型，两个网络各自拥有权重值。从生成网络开始吧。

2.4.1　生成网络

实现生成网络需要创建一个 Keras 模型，然后添加神经网络的各层。实现步骤如下。

(1) 首先确定不同超参数的取值。

```
z_size = 200
gen_filters = [512, 256, 128, 64, 1]
gen_kernel_sizes = [4, 4, 4, 4, 4]
gen_strides = [1, 2, 2, 2, 2]
gen_input_shape = (1, 1, 1, z_size)
gen_activations = ['relu', 'relu', 'relu', 'relu', 'sigmoid']
gen_convolutional_blocks = 5
```

(2) 然后创建一个输入层，以便网络接收输入。生成网络的输入是从概率潜在空间采样的一个向量。

```
input_layer = Input(shape=gen_input_shape)
```

(3) 然后添加第一个 3D 转置卷积（或 3D 反卷积）块，代码如下。

```
# 第一个 3D 转置卷积（或 3D 反卷积）块
a = Deconv3D(filters=gen_filters[0],
             kernel_size=gen_kernel_sizes[0],
             strides=gen_strides[0])(input_layer)
a = BatchNormalization()(a, training=True)
a = Activation(activation=gen_activations[0])(a)
```

(4) 再添加 4 个 3D 转置卷积（或 3D 反卷积）块，代码如下。

```
# 另外 4 个 3D 转置卷积（或 3D 反卷积）块
for i in range(gen_convolutional_blocks - 1):
    a = Deconv3D(filters=gen_filters[i + 1],
                 kernel_size=gen_kernel_sizes[i + 1],
                 strides=gen_strides[i + 1], padding='same')(a)
    a = BatchNormalization()(a, training=True)
    a = Activation(activation=gen_activations[i + 1])(a)
```

(5) 接着创建一个 Keras 模型，指明生成网络的输入和输出。

```
model = Model(inputs=input_layer, outputs=a)
```

(6) 最后将生成网络的代码包装在 `build_generator()` 函数中。

```
def build_generator():
    """
    使用如下定义的超参数，创建一个生成网络模型
    返回：生成网络
    """
    z_size = 200
    gen_filters = [512, 256, 128, 64, 1]
    gen_kernel_sizes = [4, 4, 4, 4, 4]
    gen_strides = [1, 2, 2, 2, 2]
    gen_input_shape = (1, 1, 1, z_size)
    gen_activations = ['relu', 'relu', 'relu', 'relu', 'sigmoid']
    gen_convolutional_blocks = 5
input_layer = Input(shape=gen_input_shape)

# 第一个 3D 转置卷积（或 3D 反卷积）块
a = Deconv3D(filters=gen_filters[0],
             kernel_size=gen_kernel_sizes[0],
             strides=gen_strides[0])(input_layer)
a = BatchNormalization()(a, training=True)
a = Activation(activation='relu')(a)

# 另外 4 个 3D 转置卷积（或 3D 反卷积）块
for i in range(gen_convolutional_blocks - 1):
    a = Deconv3D(filters=gen_filters[i + 1],
                 kernel_size=gen_kernel_sizes[i + 1],
                 strides=gen_strides[i + 1], padding='same')(a)
    a = BatchNormalization()(a, training=True)
    a = Activation(activation=gen_activations[i + 1])(a)

gen_model = Model(inputs=input_layer, outputs=a)

gen_model.summary()
return gen_model
```

这样就创建好了生成网络的 Keras 模型。下面为判别网络创建一个 Keras 模型。

2.4.2 判别网络

同样，实现判别网络也需要创建一个 Keras 模型，并添加神经网络的各层。实现步骤如下。

(1) 首先确定不同超参数的取值。

```
dis_input_shape = (64, 64, 64, 1)
dis_filters = [64, 128, 256, 512, 1]
dis_kernel_sizes = [4, 4, 4, 4, 4]
dis_strides = [2, 2, 2, 2, 1]
dis_paddings = ['same', 'same', 'same', 'same', 'valid']
dis_alphas = [0.2, 0.2, 0.2, 0.2, 0.2]
dis_activations = ['leaky_relu', 'leaky_relu', 'leaky_relu',
                   'leaky_relu', 'sigmoid']
dis_convolutional_blocks = 5
```

(2) 然后创建一个输入层，以便网络接收输入。判别网络的输入是一个形状为 64×64×64×1 的 3D 图像。

```
dis_input_layer = Input(shape=dis_input_shape)
```

(3) 接着添加第一个 3D 卷积块，代码如下。

```
# 第一个 3D 卷积块
a = Conv3D(filters=dis_filters[0],
           kernel_size=dis_kernel_sizes[0],
           strides=dis_strides[0],
           padding=dis_paddings[0])(dis_input_layer)
a = BatchNormalization()(a, training=True)
a = LeakyReLU(alphas[0])(a)
```

(4) 再添加 4 个 3D 卷积块，代码如下。

```
# 另外 4 个 3D 卷积块
for i in range(dis_convolutional_blocks - 1):
    a = Conv3D(filters=dis_filters[i + 1],
               kernel_size=dis_kernel_sizes[i + 1],
               strides=dis_strides[i + 1],
               padding=dis_paddings[i + 1])(a)
    a = BatchNormalization()(a, training=True)
    if dis_activations[i + 1] == 'leaky_relu':
        a = LeakyReLU(dis_alphas[i + 1])(a)
    elif dis_activations[i + 1] == 'sigmoid':
        a = Activation(activation='sigmoid')(a)
```

(5) 接着创建一个 Keras 模型，指明判别网络的输入和输出。

```
dis_model = Model(inputs=dis_input_layer, outputs=a)
```

(6) 最后将判别网络的代码包装成函数，如下所示。

```
def build_discriminator():
    """
    使用如下定义的超参数，创建一个判别网络模型
    返回：判别网络
    """
```

2

```
    dis_input_shape = (64, 64, 64, 1)
    dis_filters = [64, 128, 256, 512, 1]
    dis_kernel_sizes = [4, 4, 4, 4, 4]
    dis_strides = [2, 2, 2, 2, 1]
    dis_paddings = ['same', 'same', 'same', 'same', 'valid']
    dis_alphas = [0.2, 0.2, 0.2, 0.2, 0.2]
    dis_activations = ['leaky_relu', 'leaky_relu', 'leaky_relu',
                       'leaky_relu', 'sigmoid']
    dis_convolutional_blocks = 5

    dis_input_layer = Input(shape=dis_input_shape)

    # 第一个3D卷积块
    a = Conv3D(filters=dis_filters[0],
               kernel_size=dis_kernel_sizes[0],
               strides=dis_strides[0],
               padding=dis_paddings[0])(dis_input_layer)
    a = BatchNormalization()(a, training=True)
    a = LeakyReLU(dis_alphas[0])(a)

    # 另外4个3D卷积块
    for i in range(dis_convolutional_blocks - 1):
        a = Conv3D(filters=dis_filters[i + 1],
                   kernel_size=dis_kernel_sizes[i + 1],
                   strides=dis_strides[i + 1],
                   padding=dis_paddings[i + 1])(a)
        a = BatchNormalization()(a, training=True)
        if dis_activations[i + 1] == 'leaky_relu':
            a = LeakyReLU(dis_alphas[i + 1])(a)
        elif dis_activations[i + 1] == 'sigmoid':
            a = Activation(activation='sigmoid')(a)

    dis_model = Model(inputs=dis_input_layer, outputs=a)
    print(dis_model.summary())
return dis_model
```

这样就创建好了判别网络的Keras模型。下面开始训练3D-GAN。

2.5 训练3D-GAN

训练3D-GAN和训练普通GAN类似。首先锁定生成网络，并使用生成的图像和真实图像一起训练判别网络。然后锁定判别网络，训练生成网络。将该过程重复多轮。一次迭代中会依次训练判别网络和生成网络。3D-GAN的训练是一种端到端的训练。下面详述这些步骤。

2.5.1 训练两个网络

训练3D-GAN的步骤如下。

(1) 首先指明训练所需的不同超参数的取值，如下所示。

```
gen_learning_rate = 0.0025
dis_learning_rate = 0.00001
beta = 0.5
batch_size = 32
z_size = 200
DIR_PATH = 'Path to the 3DShapenets dataset directory'
generated_volumes_dir = 'generated_volumes'
log_dir = 'logs'
```

2

(2) 然后创建两个网络并编译，如下所示。

```
# 创建实例
generator = build_generator()
discriminator = build_discriminator()

# 确定优化器
gen_optimizer = Adam(lr=gen_learning_rate, beta_1=beta)
dis_optimizer = Adam(lr=dis_learning_rate, beta_1=0.9)

# 编译网络
generator.compile(loss="binary_crossentropy", optimizer="adam")
discriminator.compile(loss='binary_crossentropy', optimizer=dis_optimizer)
```

这里使用 Adam 优化器作为优化算法。第一步已经设置了 Adam 优化器的超参数。

(3) 接着创建对抗模型并编译。

```
discriminator.trainable = False
adversarial_model = Sequential()
adversarial_model.add(generator)
adversarial_model.add(discriminator)
adversarial_model.compile(loss="binary_crossentropy",
optimizer=Adam(lr=gen_learning_rate, beta_1=beta))
```

(4) 然后提取并加载训练所需的所有 airplane 图像。

```
def getVoxelsFromMat(path, cube_len=64):
    voxels = io.loadmat(path)['instance']
    voxels = np.pad(voxels, (1, 1), 'constant', constant_values=(0, 0))
    if cube_len != 32 and cube_len == 64:
        voxels = nd.zoom(voxels, (2, 2, 2), mode='constant', order=0)
    return voxels

def get3ImagesForACategory(obj='airplane', train=True, cube_len=64,
                           obj_ratio=1.0):
    obj_path = DIR_PATH + obj + '/30/'
    obj_path += 'train/' if train else 'test/'
    fileList = [f for f in os.listdir(obj_path) if f.endswith('.mat')]
    fileList = fileList[0:int(obj_ratio * len(fileList))]
    volumeBatch = np.asarray(
        [getVoxelsFromMat(obj_path + f, cube_len) for f in fileList],
        dtype=np.bool)
    return volumeBatch

volumes = get3ImagesForACategory(obj='airplane', train=True, obj_ratio=1.0)
volumes = volumes[..., np.newaxis].astype(np.float)
```

(5) 添加 TensorBoard 回调函数，并向其添加生成网络和判别网络。

```
tensorboard = TensorBoard(log_dir="{}/{}".format(log_dir, time.time()))
tensorboard.set_model(generator)
tensorboard.set_model(discriminator)
```

(6) 添加一个循环，进行特定轮数的训练。

```
for epoch in range(epochs):
    print("Epoch:", epoch)

    # 创建两个列表，用于存储损失
    gen_losses = []
    dis_losses = []
```

(7) 在第一个循环内部再添加一个循环，运行特定数量的批次。

```
number_of_batches = int(volumes.shape[0] / batch_size)
print("Number of batches:", number_of_batches)
for index in range(number_of_batches):
    print("Batch:", index + 1)
```

(8) 然后从真实图像集中采样一个批次的图像，同时从正态分布中采样一个批次的噪声向量。噪声向量的形状应为(1, 1, 1, 200)。

```
z_sample = np.random.normal(
    0, 0.33, size=[batch_size, 1, 1, 1, z_size]).astype(np.float32)
volumes_batch = volumes[index * batch_size:(index + 1) * batch_size, :, :, :]
```

(9) 使用生成网络生成假图像。从 z_sample 中将一批次的噪声向量传递给生成网络，以生成一批次的假图像。

```
gen_volumes = generator.predict(z_sample,verbose=3)
```

(10) 然后使用生成网络生成的假图像和真实图像集中的一批次的真图像训练判别网络。需要将判别网络设置为“可训练”。

```
# 将判别网络设置为“可训练”
discriminator.trainable = True
# 创建假标签和真标签
labels_real = np.reshape([1] * batch_size, (-1, 1, 1, 1, 1))
labels_fake = np.reshape([0] * batch_size, (-1, 1, 1, 1, 1))
# 训练判别网络
loss_real = discriminator.train_on_batch(volumes_batch,
                                         labels_real)
loss_fake = discriminator.train_on_batch(gen_volumes,
                                         labels_fake)
# 计算判别网络的总损失
d_loss = 0.5 * (loss_real + loss_fake)
```

上面的代码训练判别网络，并计算判别网络的全部损失。

(11) 对包含生成网络和判别网络的对抗模型进行训练。

```
z = np.random.normal(0, 0.33, size=[batch_size, 1, 1, 1,
                     z_size]).astype(np.float32)

        # 训练对抗模型
        g_loss = adversarial_model.train_on_batch(
            z, np.reshape([1]*batch_size, (-1, 1, 1, 1, 1)))
```

将两个损失分别加入相应列表中。

```
gen_losses.append(g_loss)
dis_losses.append(d_loss)
```

(12) 每训练两轮，生成并保存 3D 图像。

```
if index % 10 == 0:
    z_sample2 = np.random.normal(
        0, 0.33, size=[batch_size, 1, 1, 1, z_size]).astype(np.float32)
    generated_volumes = generator.predict(z_sample2, verbose=3)
    for i, generated_volume in enumerate(generated_volumes[:5]):
        voxels = np.squeeze(generated_volume)
        voxels[voxels < 0.5] = 0.
        voxels[voxels >= 0.5] = 1.
        saveFromVoxels(voxels, "results/img_{}_{}_{}".format(epoch,index, i))
```

(13) 每轮将平均损失保存到 TensorBoard。

```
# 将损失存储到 TensorBoard
write_log(tensorboard, 'g_loss', np.mean(gen_losses), epoch)
write_log(tensorboard, 'd_loss', np.mean(dis_losses), epoch)
```

TIP

建议首先训练 100 轮以便发现代码中的问题。排查完问题后就可以进行 100 000 轮的训练了。

2.5.2 保存模型

完成训练之后，使用如下代码保存生成网络和判别网络习得的权重值。

```
"""
保存模型
"""
generator.save_weights(os.path.join(generated_volumes_dir,
                                    "generator_weights.h5"))
discriminator.save_weights(os.path.join(generated_volumes_dir,
                                        "discriminator_weights.h5"))
```

2.5.3　测试模型

如果想测试模型，需要创建生成网络和判别网络并加载习得的权重值，然后使用 predict() 方法生成估测。

```
# 创建模型
generator = build_generator()
discriminator = build_discriminator()

# 加载模型权重
generator.load_weights(os.path.join(generated_volumes_dir,
                                    "generator_weights.h5"), True)
discriminator.load_weights(os.path.join(generated_volumes_dir,
                                        "discriminator_weights.h5"), True)

# 生成 3D 图像
z_sample = np.random.normal(
    0, 0.33, size=[batch_size, 1, 1, 1, z_size]).astype(np.float32)
generated_volumes = generator.predict(z_sample, verbose=3)
```

这样就训练了 3D-GAN 的生成网络和判别网络。下面介绍超参数调优和优化超参数的各种方法。

2.5.4　损失可视化

启动 TensorBoard 服务器，将训练损失可视化，如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图（见图 2-9 和图 2-10）。

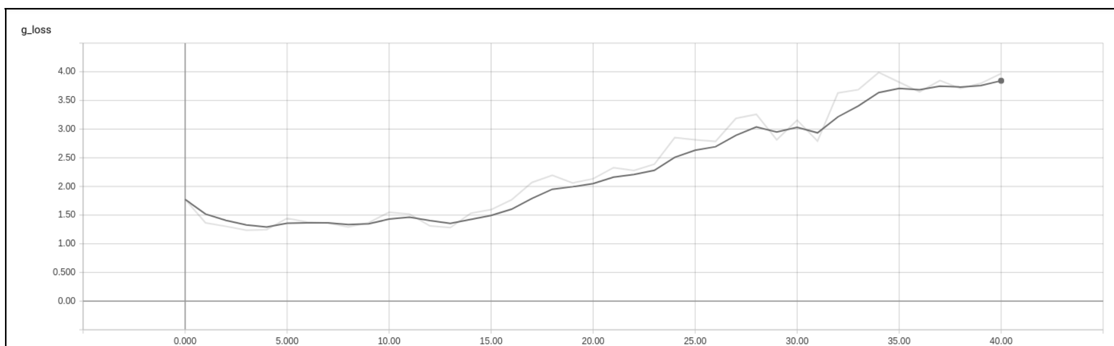


图 2-9　生成网络的损失曲线

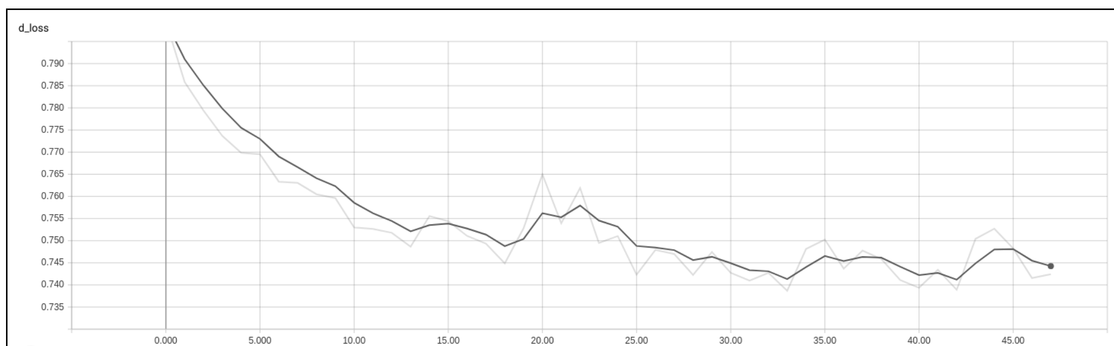


图 2-10　判别网络的损失曲线

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了；如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果；如果损失在逐渐降低，就继续训练模型。

2.5.5　图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 2-11）。

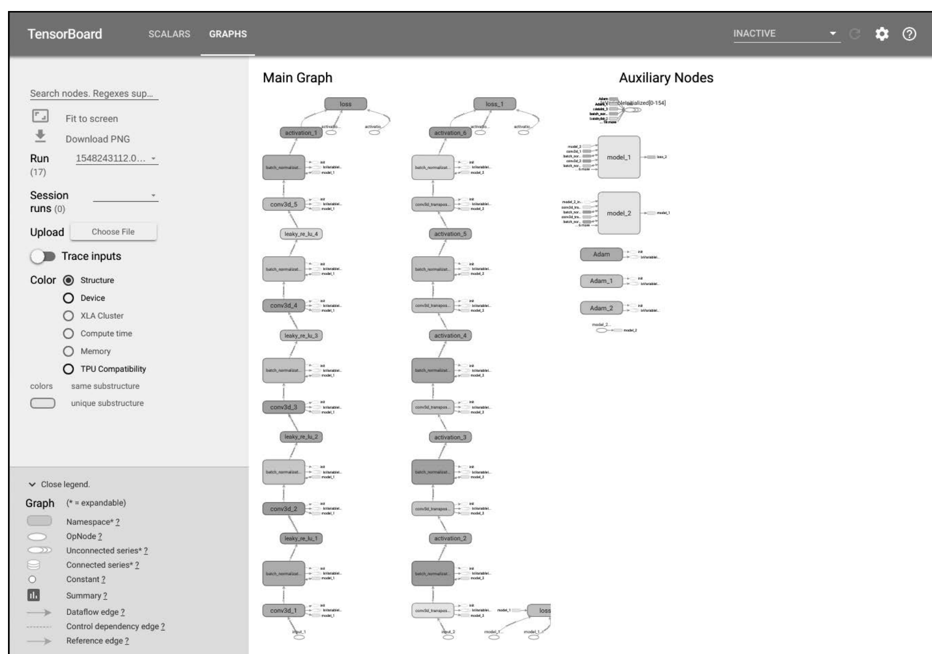


图 2-11　各图中的张量和不同运算的流

2.6　超参数优化

前面训练的模型可能不够完美，可以通过超参数调优来改进。3D-GAN 中有很多用于优化的超参数，如下所示。

- **批次规模**。尝试使用 8、16、32、64 或 128 作为批次规模。
- **轮数**。尝试训练 100 轮，然后逐渐提高到 1000 轮到 5000 轮。
- **学习速率**。这是最重要的超参数。尝试 0.1、0.001、0.0001，以及其他较小的学习速率。
- **生成网络和判别网络中各层的激活函数**：尝试使用 sigmoid、tanh、ReLU、LeakyReLU、ELU、SeLU，以及其他激活函数。
- **优化算法**。尝试使用 Adam、SGD、Adadelta、RMSProp，以及 Keras 框架中的其他优化器。
- **损失函数**。对于 3D-GAN 而言，二元交叉熵是最佳的损失函数。
- **两个网络的层数**。根据可用训练数据量的大小，尝试使用不同的网络层数。如果训练数据量足够的话，可以使用深度网络。

也可以使用 Hyperopt 或 Hyperas 等库来实现自动化超参数优化，以找到最佳的超参数组合。

2.7　3D-GAN 的实际应用

3D-GAN 在很多行业都有应用前景，如下所示。

- **制造业**。作为一种创造性工具，3D-GAN 有助于快速创建原型。它们可以启发创意，还有助于 3D 模型的模拟和可视化。
- **3D 打印**。3D-GAN 生成的 3D 图像可用于 3D 打印。人工构建 3D 模型非常耗时。
- **设计过程**。生成 3D 模型可以为某个具体流程的最终结果提供很好的预估。它们可以展现所设计东西的最终形态。
- **生成新样本**。类似于其他 GAN，3D-GAN 可用于生成图像，以训练有监督模型。

2.8　小结

本章探讨了 3D-GAN。首先介绍了 3D-GAN，以及其生成网络和判别网络的架构和配置，然后通过一系列步骤建立了项目，并介绍了如何准备数据集，随后使用 Keras 框架实现了一个 3D-GAN，并在数据集上进行了训练。本章还列出了不同的超参数选择，最后介绍了 3D-GAN 的实际应用。

下一章将介绍如何使用 cGAN 实现人脸老化。

第 3 章

使用 cGAN 实现人脸老化

cGAN 是 GAN 模型的一种扩展，可以生成符合某些条件或特征的图像，并且效果好于普通 GAN。本章实现一个 cGAN，训练之后可以自动进行人脸老化。该 cGAN 最初是由 Grigory Antipov、Moez Baccouche 和 Jean-Luc Dugelay 在论文“Face Aging With Conditional Generative Adversarial Networks”中提出的。

本章讨论以下主题。

- 人脸老化 cGAN 简介
- 项目创建
- 数据准备
- cGAN 的 Keras 实现
- 训练 cGAN
- 模型评估以及超参数调优
- 人脸老化的实际应用

3.1 人脸老化 cGAN 简介

前面实现过不同用途的 GAN。cGAN 拓展了普通 GAN 的概念，可以控制生成网络的输出。人脸老化是在保留身份特征的情况下，改变人脸的年龄。使用其他大多数模型（包括 GAN）解决该问题时，人的外貌或者身份特征会损失 50%，因为面部表情以及修饰（比如墨镜或胡子）都没能考虑在内，而 Age-cGAN 会考虑所有这些特征。下面介绍可实现人脸老化的 cGAN。

3.1.1 理解 cGAN

cGAN 能以某种额外信息为条件，其生成网络接收额外信息 y 作为附加输入层。普通 GAN 无法控制生成图像的类别。如果向生成网络增加条件 y（可以是任何类型的数据，比如类别标签或整数）的话，就可以使用 y 来生成特定类别的图像。普通 GAN 只能学习一个类别，构建可以学习多个类别的 GAN 极其困难。而 cGAN 却能生成多峰模型，使用不同的条件生成不同的类别。

图 3-1 展示了一个 cGAN 架构。

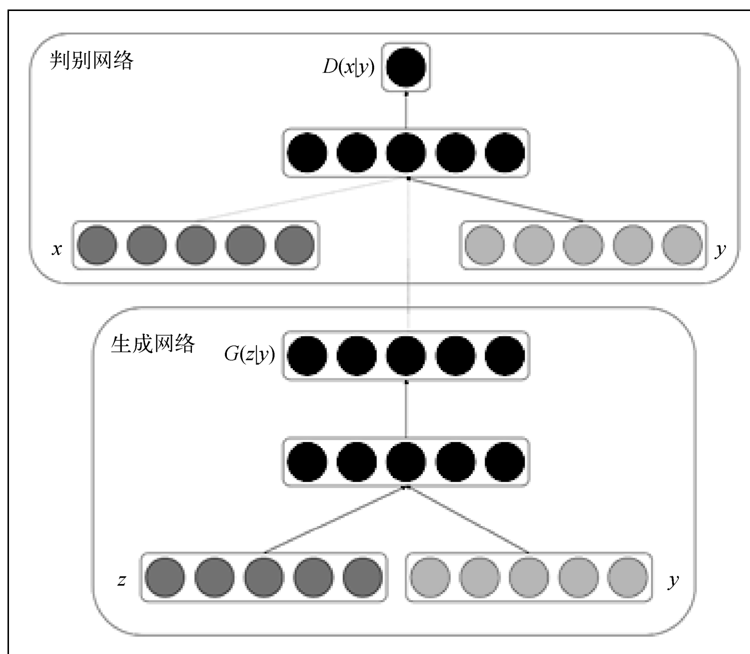


图 3-1　cGAN 架构

cGAN 的训练目标函数形式如下。

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x \mid y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z \mid y)))]$$

其中，G 是生成网络，D 是判别网络。判别网络的损失是 $\log D(x \mid y)$，生成网络的损失是 $\log(1-D(G(z \mid y)))$。给定 z 和 y，$G(z \mid y)$相当于对数据分布的建模。其中，z 是一个从正态分布中采样的 100 维先验噪声分布。

3.1.2　Age-cGAN 架构

人脸老化 cGAN 的架构稍复杂。Age-cGAN 包含 4 个网络：1 个编码网络、1 个 FaceNet、1 个生成网络和 1 个判别网络。使用编码网络可以学习具有年龄条件的人脸图像与对应潜在向量 z_0 之间的反向映射关系。FaceNet 是一个人脸识别网络，学习判别输入图像 x 和重构图像 $\tilde{x}$ 之间的差别。生成网络接收隐藏表示（由人脸图像和条件向量组成），然后生成图像。判别网络用于区分真实图像和假图像。

cGAN 存在的问题是无法学习具有特征 y 的输入图像 x 到潜在向量 z 的反向映射。使用编码网络可以解决该问题。可以训练编码网络对输入图像 x 反向映射的结果进行近似。下面介绍 Age-cGAN

涉及的各个网络。

1. 编码网络

编码网络用于生成给定图像的潜在向量。它接收维度为`(64, 64, 3)`的图像，将其转换成100维。编码网络是深度卷积神经网络。该网络包含4个卷积块和2个全连接层。每个卷积块包含一个卷积层、一个批归一化层和一个激活函数。对于每个卷积块，除了第一个卷积层，其余每个卷积层后面都有一个批归一化层。3.4节会介绍编码网络的配置。

2. 生成网络

生成网络用于生成维度为`(64, 64, 3)`的图像。它读取一个100维潜在向量以及额外信息 y，然后试图生成逼真的图像。生成网络也是深度卷积神经网络，由全连接层、上采样层和卷积层组成。它读取两个输入值：一个噪声向量和一个条件值。条件值就是向生成网络提供的附加信息，在Age-cGAN中是年龄。3.3节会介绍生成网络的配置。

3. 判别网络

判别网络用于判断给定图像的真假，通过一系列下采样层和一些分类层处理图像来实现，即它估测该图像是真是假。判别网络也是深度卷积网络，包含一些卷积块。除了第一个卷积块没有批归一化层之外，其余卷积块都包含一个卷积层、一个批归一化层和一个激活函数。3.4 节会介绍判别网络的配置。

4. 人脸识别网络

人脸识别网络旨在通过给定图像识别一个人的身份特征。本章会使用预训练的Inception-ResNet-2模型，但是去掉了其全连接层。Keras 拥有很不错的预训练模型库。也可以使用 Inception 或者ResNet-50等网络来进行试验。预训练的Inception-ResNet-2返回给定图像对应的嵌入。后面的计算会用到从真实图像和重构图像中抽取的嵌入，因为需要计算两者之间的欧氏距离。3.4 节会深入探讨人脸识别网络。

3.1.3 Age-cGAN 的训练阶段

Age-cGAN有多个训练阶段。前面提过，Age-cGAN由4个网络组成，通过如下3个阶段进行训练。

(1) **cGAN 训练**：该阶段训练生成网络和判别网络。
(2) **潜在向量初步近似**：该阶段训练编码网络。
(3) **潜在向量优化**：该阶段同时训练编码网络和生成网络。

图3-2展示了Age-cGAN的各个训练阶段。

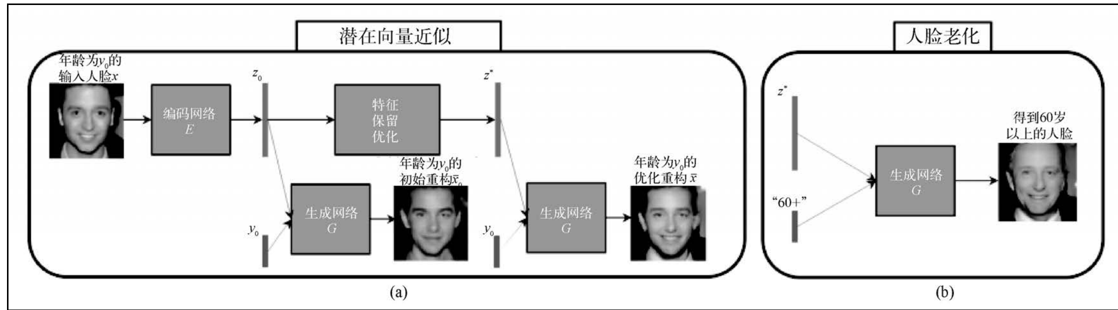


图 3-2　Age-cGAN 的各个训练阶段

下面介绍 Age-cGAN 的各个阶段。

1 cGAN 训练

该阶段训练生成网络和判别网络。训练完成后，生成网络可以生成模糊的人脸图像。该阶段和训练普通 GAN 类似，同时训练两个网络。

- **训练目标函数**

cGAN 的训练目标函数形式如下。

$$\min_{\theta_G}\max_{\theta_G} v(\theta_G,\theta_D) = \mathbb{E}_{x,y\sim p_{\text{data}}}[\log D(x,y)] \\ +\mathbb{E}_{z\sim p_z(z),\tilde{y}\sim p_y}[\log(1-D(G(z,\tilde{y}),\tilde{y}))] \tag{1}$$

cGAN 的训练涉及优化 $v(\theta_G,\theta_D)$ 函数。可以把 cGAN 的训练看作极小极大化的博弈，其中生成网络和判别网络同时进行训练。在上面的公式中，θ_G 表示生成网络的参数，θ_D 表示判别网络的参数，$\log D(x,y)$ 表示判别网络的损失，$\log(1-D(G(z,\tilde{y}),\tilde{y}))$ 表示生成网络的损失，p_{data} 是所有可能图像的概率分布。

2. 潜在向量初步近似

潜在向量初步近似是一种对潜在向量进行近似以优化人脸图像重构的方法。对潜在向量进行近似需要使用编码网络，编码网络使用生成的图像和真实图像进行训练。训练完成后，编码网络可以从习得的概率分布中生成潜在向量。编码网络的训练目标函数是欧氏距离损失。

3. 潜在向量优化

潜在向量优化阶段同时对编码网络和生成网络进行优化。公式如下。

$$z^*IP = \arg\min_z \|FR(x)-FR(\tilde{x})\|_{L2}$$

其中 *FR* 是人脸识别网络。该公式表明真实图像和重构图像之间的欧氏距离应该最小化。此阶段试图对该距离进行最小化，以最大程度保留身份特征。

3.2 创建项目

首先需要克隆包含本书所有章节完整代码的仓库。其中目录 Chapter03 包含本章的完整代码。执行如下命令来创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter03。

```
cd Chapter03
```

(3) 然后为本项目创建一个 Python 虚拟环境。

```
virtualenv venv
virtualenv venv -p python3 # 创建使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 启用新创建的虚拟环境。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 接着使用如下命令安装 requirements.txt 文件中列出的所有库。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

至此，成功创建了项目并安装了所需的依赖程序。下面准备数据集。

3.3 准备数据

本章使用 Wiki-Cropped 数据集，其中包含了 64 328 张不同人的面部图像。其作者还提供了一个包含剪裁后图像的数据集，所以本项目不需要剪裁图像了。



论文“Deep expectation of real and apparent age from a single image without facial landmarks”的作者从维基百科爬取了这些图像，并制成数据集供学术研究使用。如想商用，请发送邮件到 rrothe@vision.ee.ethz.ch 联系作者。

该数据集的下载地址为：https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/。下载后请将所有压缩文件放在 Age-cGAN 项目的目录中。

下载及提取数据集的步骤如下。

3.3.1　下载数据集

执行下面的命令，下载只包含剪裁后的人脸图像数据集。

```
# 首先访问数据目录
cd data

# 维基百科数据集：仅下载面部图像
wget
https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/static/wiki_crop.tar
```

3.3.2　提取数据集

数据集下载完成之后，手动提取数据文件夹里的文件，或者执行如下命令来提取文件。

```
# 访问数据目录 cd data
cd data

# 提取 wiki_crop.tar
tar -xvf wiki_crop.tar
```

wiki_crop.tar 文件包含 62 328 张图像，以及一个包含所有标签的 wiki.mat 文件。可以使用 scipy.io 库中的 `loadmat` 方法在 Python 中加载.mat 文件。使用如下代码加载提取出来的.mat 文件。

```
def load_data(wiki_dir, dataset='wiki'):
    # 加载 wiki.mat 文件
    meta = loadmat(os.path.join(wiki_dir, "{}.mat".format(dataset)))

    # 加载文件列表
    full_path = meta[dataset][0, 0]["full_path"][0]

    # 包含 Matlab 日期序号的列表
    dob = meta[dataset][0, 0]["dob"][0]

    # 包含照片拍摄年份的列表
    photo_taken = meta[dataset][0, 0]["photo_taken"][0] # year

    # 对于每个出生日期数据，计算其在照片拍摄年份所对应的年龄
    age = [calculate_age(photo_taken[i], dob[i]) for i in range(len(dob))]
```

```
    # 创建由元组构成的列表，其中每个元组包含一个图像路径和一个年龄
    images = []
    age_list = []
    for index, image_path in enumerate(full_path):
        images.append(image_path[0])
        age_list.append(age[index])

    # 返回包含所有图像的列表，以及一个包含对应年龄的列表
    return images, age_list
```

photo_taken 变量是一个年份列表，dob 是列表中照片对应的 Matlab 日期序号。可以通过日期序号和照片拍摄时间来计算人的年龄，如以下代码所示。

```
def calculate_age(taken, dob):
    birth = datetime.fromordinal(max(int(dob) - 366, 1))

    # 假设照片是在当年的年中拍摄的
    if birth.month < 7:
        return taken - birth.year
    else:
        return taken - birth.year - 1
```

下载并提取数据集后，下面介绍 Age-cGAN 的 Keras 实现。

3.4 Age-cGAN 的 Keras 实现

类似于普通 GAN，cGAN 不难实现。Keras 为编写复杂的生成对抗网络提供了足够的灵活性。下面实现 cGAN 中的生成网络、判别网络和编码网络。从编码网络开始。

编写实现代码之前，首先创建一个 Python 文件 main.py 并导入核心模块，如下所示。

```
import math
import os
import time
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from keras import Input, Model
from keras.applications import InceptionResNetV2
from keras.callbacks import TensorBoard
from keras.layers import Conv2D, Flatten, Dense, BatchNormalization, \
    Reshape, concatenate, LeakyReLU, Lambda, \
        K, Conv2DTranspose, Activation, UpSampling2D, Dropout
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras_preprocessing import image
from scipy.io import loadmat
```

3.4.1　编码网络

编码网络是 CNN，将图像（x）编码为潜在向量（z）或潜在向量表示。首先使用 Keras 框架实现编码网络。

实现编码网络的步骤如下。

(1) 首先创建一个输入层。

```
input_layer = Input(shape=(64, 64, 3))
```

(2) 然后添加第一个卷积块，包含一个 2D 卷积层和一个激活函数。配置如下。

- **过滤器数量**：32
- **卷积核大小**：5
- **步长**：2
- **填充方式**：same
- **激活函数**：alpha 值为 0.2 的 LeakyReLU

```
# 第 1 个卷积块
enc = Conv2D(filters=32, kernel_size=5, strides=2,
             padding='same')(input_layer)
enc = LeakyReLU(alpha=0.2)(enc)
```

(3) 然后再添加 3 个卷积块，每个卷积块包含 1 个 2D 卷积层、1 个批归一化层和 1 个激活函数。配置如下。

- **过滤器数量**：64，128，256
- **卷积核大小**：5，5，5
- **步长**：2，2，2
- **填充方式**：same，same，same
- **批归一化**：每个卷积层后面都会使用一个批归一化层
- **激活函数**：LealyReLU，LeakyReLU，LeakyReLU（alpha 值都是 0.2）

```
# 第 2 个卷积块
enc = Conv2D(filters=64, kernel_size=5, strides=2,
             padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)

# 第 3 个卷积块
enc = Conv2D(filters=128, kernel_size=5, strides=2,
             padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)
```

```
# 第 4 个卷积块
enc = Conv2D(filters=256, kernel_size=5, strides=2,
             padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)
```

(4) 接着将最后一个卷积块的输出扁平化，如下所示。

```
# 扁平化层
enc = Flatten()(enc)
```



将 *n* 维张量转换成一维张量（数列）的操作称为**扁平化**。

(5) 然后添加一个全连接层，以及一个批归一化层和一个激活函数。配置如下。

- **单元（结点）**：2096
- **是否使用批归一化**：是
- **激活函数**：alpha 值为 0.2 的 LeakyReLU

```
# 第 1 个全连接层
enc = Dense(4096)(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)
```

(6) 接着添加第 2 个全连接层，配置如下。

- **单元（结点）**：100
- **激活函数**：无

```
# 第 2 个全连接层
enc = Dense(100)(enc)
```

(7) 最后，创建一个 Keras 模型，并指明编码网络的输入和输出。

```
# 创建模型
model = Model(inputs=[input_layer], outputs=[enc])
```

编码网络的完整代码如下：

```
def build_encoder():
    """
    编码网络
    """
    input_layer = Input(shape=(64, 64, 3))

    # 第 1 个卷积块
    enc = Conv2D(filters=32, kernel_size=5, strides=2,
                 padding='same')(input_layer)
    enc = LeakyReLU(alpha=0.2)(enc)
```

```
# 第 2 个卷积块
enc = Conv2D(filters=64, kernel_size=5, strides=2, padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)

# 第 3 个卷积块
enc = Conv2D(filters=128, kernel_size=5, strides=2,
             padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)

# 第 4 个卷积块
enc = Conv2D(filters=256, kernel_size=5, strides=2,
             padding='same')(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)

# 扁平化层
enc = Flatten()(enc)

# 第 1 个全连接层
enc = Dense(4096)(enc)
enc = BatchNormalization()(enc)
enc = LeakyReLU(alpha=0.2)(enc)

# 第 2 个全连接层
enc = Dense(100)(enc)

# 创建模型
model = Model(inputs=[input_layer], outputs=[enc])
return model
```

这样就创建好了编码网络的 Keras 模型。下面创建生成网络的 Keras 模型。

3.4.2　生成网络

生成网络是 CNN，接收 100 维向量 z，输出维度为`(64, 64, 3)`的图像。下面使用 Keras 框架实现生成网络。

实现生成网络的步骤如下。

(1) 首先创建生成网络的两个输入层。

```
latent_dims = 100
num_classes = 6

# 向量 z 的输入层
input_z_noise = Input(shape=(latent_dims, ))

# 条件变量的输入层
input_label = Input(shape=(num_classes, ))
```

(2) 然后将两个输入张量沿通道维度进行拼接，如下所示。

```
x = concatenate([input_z_noise, input_label])
```

这一步会拼接张量。

(3) 再添加一个全连接块，配置如下。

- 单元（结点）：2048
- 输入维度：106
- 激活函数：alpha 值为 0.2 的 LeakyReLU
- 随机失活系数：0.2

```
x = Dense(2048, input_dim=latent_dims+num_classes)(x)
x = LeakyReLU(alpha=0.2)(x)
x = Dropout(0.2)(x)
```

(4) 接着添加第二个全连接块，配置如下。

- 单元（结点）：16 384
- 是否使用批归一化：是
- 激活函数：alpha 值为 0.2 的 LeakyReLU
- 随机失活系数：0.2

```
x = Dense(256 * 8 * 8)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
x = Dropout(0.2)(x)
```

(5) 然后改变全连接层输出张量的形状，将其变成维度为`(8, 8, 256)`的三维张量。

```
x = Reshape((8, 8, 256))(x)
```

该层会生成维度是`(batch_size, 8, 8, 256)`的张量。

(6) 接着添加一个上采样块，包含一个上采样层、一个 2D 卷积层和一个批归一化层。配置如下。

- 上采样系数：`(2, 2)`
- 过滤器数量：128
- 卷积核大小：5
- 填充方式：same
- 是否使用批归一化：是，其 `momentum` 值为 0.8
- 激活函数：alpha 值为 0.2 的 LeakyReLU

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(filters=128, kernel_size=5, padding='same')(x)
x = BatchNormalization(momentum=0.8)(x)
x = LeakyReLU(alpha=0.2)(x)
```



Upsampling2D 操作将张量的行重复 x 次，将其列重复 y 次。

(7) 再添加一个上采样块（和上一个类似），如以下代码所示。其配置和上一个块类似，但卷积层中过滤器的数量为 64。

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(filters=64, kernel_size=5, padding='same')(x)
x = BatchNormalization(momentum=0.8)(x)
x = LeakyReLU(alpha=0.2)(x)
```

(8) 然后添加最后一个上采样块。其配置和上一个类似，但卷积层中过滤器的数量为 3，并且没有使用批归一化。

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(filters=3, kernel_size=5, padding='same')(x)
x = Activation('tanh')(x)
```

(9) 最后，创建 Keras 模型，并指明生成网络的输入和输出。

```
model = Model(inputs=[input_z_noise, input_label], outputs=[x])
```

生成网络的完整代码如下。

```
def build_generator():
    """
    使用如下定义的超参数，创建一个生成网络模型
    返回：生成网络模型
    """
    latent_dims = 100
    num_classes = 6

    input_z_noise = Input(shape=(latent_dims,))
    input_label = Input(shape=(num_classes,))

    x = concatenate([input_z_noise, input_label])

    x = Dense(2048, input_dim=latent_dims + num_classes)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.2)(x)

    x = Dense(256 * 8 * 8)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.2)(x)

    x = Reshape((8, 8, 256))(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(filters=128, kernel_size=5, padding='same')(x)
    x = BatchNormalization(momentum=0.8)(x)
```

```
    x = LeakyReLU(alpha=0.2)(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(filters=64, kernel_size=5, padding='same')(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(filters=3, kernel_size=5, padding='same')(x)
    x = Activation('tanh')(x)

    model = Model(inputs=[input_z_noise, input_label], outputs=[x])
    return model
```

这样就创建好了生成网络。下面编写判别网络的代码。

3.4.3 判别网络

本项目使用的判别网络是 CNN。下面使用 Keras 框架来实现。

实现判别网络的步骤如下。

(1) 首先创建两个输入层，因为该判别网络需要处理两个输入。

```
# 确定超参数
# 输入图像的形状
input_shape = (64, 64, 3)
# 输入条件变量的形状
label_shape = (6,)

# 两个输入层
image_input = Input(shape=input_shape)
label_input = Input(shape=label_shape)
```

(2) 然后添加一个 2D 卷积块（Conv2D+激活函数），其配置如下。

- 过滤器数量：64
- 卷积核大小：3
- 步长：2
- 填充方式：same
- 激活函数：alpha 值为 0.2 的 LeakyReLU

```
x = Conv2D(64, kernel_size=3, strides=2,
           padding='same')(image_input)
x = LeakyReLU(alpha=0.2)(x)
```

(3) 接着将 label_input 扩展成形状为`(32, 32, 6)`的张量。

```
label_input1 = Lambda(expand_label_input)(label_input)
```

expand_label_input 函数如下所示。

```
# expand_label_input 函数
def expand_label_input(x):
    x = K.expand_dims(x, axis=1)
    x = K.expand_dims(x, axis=1)
    x = K.tile(x, [1, 32, 32, 1])
    return x
```

该函数将张量的维度从(6,)转换成(32, 32, 6)。

(4) 然后将转换后的标签张量和上一个卷积层的输出沿通道维度进行拼接，如下所示。

```
x = concatenate([x, label_input1], axis=3)
```

(5) 添加一个卷积块（2D 卷积层+批归一化+激活函数），配置如下。

- 过滤器数量：128
- 卷积核大小：3
- 步长：2
- 填充方式：same
- 是否使用批归一化：是
- 激活函数：alpha 值为 0.2 的 LeakyReLU

```
x = Conv2D(128, kernel_size=3, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
```

(6) 再添加两个卷积块，如下所示。

```
x = Conv2D(256, kernel_size=3, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(512, kernel_size=3, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
```

(7) 然后添加一个扁平化层。

```
x = Flatten()(x)
```

(8) 接着添加一个全连接层（分类层），输出概率值。

```
x = Dense(1, activation='sigmoid')(x)
```

(9) 最后，创建一个 Keras 模型，并指明判别网络的输入和输出。

```
model = Model(inputs=[image_input, label_input], outputs=[x])
```

判别网络的完整代码如下。

```
def build_discriminator():
    """
    使用如下定义的超参数，创建一个判别网络模型
    返回：判别网络模型
    """
    input_shape = (64, 64, 3)
    label_shape = (6,)
    image_input = Input(shape=input_shape)
    label_input = Input(shape=label_shape)

    x = Conv2D(64, kernel_size=3, strides=2, padding='same')(image_input)
    x = LeakyReLU(alpha=0.2)(x)

    label_input1 = Lambda(expand_label_input)(label_input)
    x = concatenate([x, label_input1], axis=3)
    x = Conv2D(128, kernel_size=3, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(256, kernel_size=3, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(512, kernel_size=3, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Flatten()(x)
    x = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=[image_input, label_input], outputs=[x])
    return model
```

这样就创建好了编码网络、生成网络和判别网络。下面把它们整合起来，训练网络。

3.5 训练 cGAN

训练人脸老化 cGAN 分 3 步。

(1) 训练 cGAN。
(2) 潜在向量初步近似。
(3) 优化潜在向量。

下面依次介绍这些步骤。

3.5.1 训练 cGAN

首先训练生成网络和判别网络，步骤如下。

(1) 首先指定训练所需参数。

```
# 定义超参数
data_dir = "/path/to/dataset/directory/"
wiki_dir = os.path.join(data_dir, "wiki_crop")
epochs = 500
batch_size = 128
image_shape = (64, 64, 3)
z_shape = 100
TRAIN_GAN = True
TRAIN_ENCODER = False
TRAIN_GAN_WITH_FR = False
fr_image_shape = (192, 192, 3)
```

(2) 接着定义训练要用的优化器。这里使用 Keras 的 Adam 优化器。初始化优化器，代码如下所示。

```
# 定义优化器
# 判别网络使用的优化器
dis_optimizer = Adam(lr=0.0002, beta_1=0.5, beta_2=0.999,
                     epsilon=10e-8)

# 生成网络使用的优化器
gen_optimizer = Adam(lr=0.0002, beta_1=0.5, beta_2=0.999,
                     epsilon=10e-8)

# 对抗网络使用的优化器
adversarial_optimizer = Adam(lr=0.0002, beta_1=0.5, beta_2=0.999,
                             epsilon=10e-8)
```

对于所有优化器，设置学习速率为 0.0002、`beta_1` 值为 0.5、`beta_2` 值为 0.999、epsilon 值设为 10e-8。

(3) 然后加载生成网络和判别网络并编译。在 Keras 中，网络必须经过编译才能进行训练。

```
# 构建并编译判别网络
discriminator = build_discriminator()
discriminator.compile(loss=['binary_crossentropy'],
                      optimizer=dis_optimizer)
# 构建并编译生成网络
generator = build_generator1()
generator.compile(loss=['binary_crossentropy'],
                  optimizer=gen_optimizer)
```

编译网络时，使用 `binary_crossentropy` 作为损失函数。

(4) 然后构建对抗模型并编译，如下所示。

```
# 构建对抗模型并编译
discriminator.trainable = False
input_z_noise = Input(shape=(100,))
input_label = Input(shape=(6,))
```

```
recons_images = generator([input_z_noise, input_label])
valid = discriminator([recons_images, input_label])
adversarial_model = Model(inputs=[input_z_noise, input_label],
                          outputs=[valid])
adversarial_model.compile(loss=['binary_crossentropy'],
                          optimizer=gen_optimizer)
```

编译对抗模型时，使用 `binary_crossentropy` 作为损失函数，使用 `gen_optimizer` 作为优化器。

(5) 接着添加 TensorBoard 以存储损失，如下所示。

```
tensorboard = TensorBoard(log_dir="logs/{}".format(time.time()))
tensorboard.set_model(generator)
tensorboard.set_model(discriminator)
```

(6) 然后使用 3.3 节定义的 `load_data()` 函数加载全部图像[①]。

```
images, age_list = load_data(wiki_dir=wiki_dir, dataset="wiki")
```

(7) 接着将年龄数值转换成年龄类别，如下所示。

```
# 将年龄转换为类别变量
age_cat = age_to_category(age_list)
```

`age_to_category` 函数的定义如下。

```
# 该函数将年龄转换为相应的类别
def age_to_category(age_list):
    age_list1 = []

    for age in age_list:
        if 0 < age <= 18:
            age_category = 0
        elif 18 < age <= 29:
            age_category = 1
        elif 29 < age <= 39:
            age_category = 2
        elif 39 < age <= 49:
            age_category = 3
        elif 49 < age <= 59:
            age_category = 4
        elif age >= 60:
            age_category = 5

        age_list1.append(age_category)
    return age_list1
```

`age_cat` 的输出如下所示。

① 这里加载的实际上应该是图像的路径。——译者注

```
[1, 2, 4, 2, 3, 4, 2, 5, 5, 1, 3, 2, 1, 1, 2, 1, 2, 2, 1, 5, 4 ,
.............]
```

将年龄类别转换成独热编码向量。

```
# 并且将年龄类别转换为独热编码向量
final_age_cat = np.reshape(np.array(age_cat), [len(age_cat), 1])
classes = len(set(age_cat))
y = to_categorical(final_age_cat, num_classes=len(set(age_cat)))
```

年龄类别转换成独热编码向量之后，*y* 值应如下所示。

```
[[0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 ...
 [0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]]
```

y 的形状应为`(total_values, 5)`。

(8) 然后加载所有图像，并创建包含所有图像的 ndarray。

```
# 读取所有图像，并创建一个 ndarray
loaded_images = load_images(wiki_dir, images, (image_shape[0], image_shape[1]))
```

`load_images()`函数的定义如下。

```
def load_images(data_dir, image_paths, image_shape):
    images = None

    for i, image_path in enumerate(image_paths):
        print()
        try:
            # 加载图像
            loaded_image = image.load_img(
                os.path.join(data_dir, image_path), target_size=image_shape)

            # 将 PIL 图像转换为 NumPy ndarray
            loaded_image = image.img_to_array(loaded_image)

            # 添加另外一个维度（即批次的维度）
            loaded_image = np.expand_dims(loaded_image, axis=0)

            # 将所有图像拼接为一个张量
            if images is None:
                images = loaded_image
            else:
                images = np.concatenate([images, loaded_image], axis=0)
        except Exception as e:
            print("Error:", i, e)

    return images
```

loaded_images 包含的值如下所示。

```
[[[[ 97. 122. 178.]
 [ 98. 123. 179.]
 [ 99. 124. 180.]
 ...
 [ 97. 124. 179.]
 [ 96. 123. 178.]
 [ 95. 122. 177.]]
...
[[216. 197. 203.]
 [217. 198. 204.]
 [218. 199. 205.]
 ...
 [ 66. 75. 90.]
 [110. 127. 171.]
 [ 89. 115. 172.]]]
 [[[122. 140. 152.]
 [115. 133. 145.]
 [ 95. 113. 123.]
 ...
 [ 41. 73. 23.]
 [ 38. 77. 22.]
 [ 38. 77. 22.]]
[[ 53. 80. 63.]
 [ 47. 74. 57.]
 [ 45. 72. 55.]
 ...
 [ 34. 66...
```

(9) 接着创建一个 for 循环，其运行次数应与指定的训练轮数一致，如下所示。

```
for epoch in range(epochs):
    print("Epoch:{}".format(epoch))

    gen_losses = []
    dis_losses = []

    number_of_batches = int(len(loaded_images) / batch_size)
    print("Number of batches:", number_of_batches)
```

(10) 然后在该循环内部再创建一个循环，其运行次数应与 num_batches 一致，如下所示。

```
for index in range(number_of_batches):
    print("Batch:{}".format(index + 1))
```

训练判别网络和对抗网络的完整代码都会放在该循环里面。

(11) 接着从真实数据集中采样一批次图像，并采样其对应的独热编码年龄向量。

```
images_batch = loaded_images[index * batch_size:(index + 1) * batch_size]
images_batch = images_batch / 127.5 - 1.0
images_batch = images_batch.astype(np.float32)

y_batch = y[index * batch_size:(index + 1) * batch_size]
```

3

image_batch 的形状应为(batch_size, 64, 64, 3)，y_batch 的形状应为(batch_size, 6)。

(12) 从高斯分布中采样一批次的噪声向量，如下所示。

```
z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))
```

(13) 然后使用生成网络生成假图像。请注意，生成网络还未训练过。

```
initial_recon_images = generator.predict_on_batch([z_noise, y_batch])
```

生成网络接收两个输入，z_noise 和 y_batch，分别是第(11)步和第(12)步创建的。

(14) 下面使用真实图像和假图像训练判别网络。

```
d_loss_real = discriminator.train_on_batch([images_batch, y_batch], real_labels)
d_loss_fake = discriminator.train_on_batch(
    [initial_recon_images, y_batch], fake_labels)
```

上述代码使用一批次的图像训练判别网络。判别网络每一步会使用一批次的采样进行训练。

(15) 然后训练对抗网络。锁定判别网络，只训练生成网络。

```
# 再次从高斯（正态）分布中采样一批次的噪声向量
z_noise2 = np.random.normal(0, 1, size=(batch_size, z_shape))

# 采样一批次的随机年龄数值
random_labels = np.random.randint(0, 6, batch_size).reshape(-1, 1)

# 将随机年龄数值转换成独热编码
random_labels = to_categorical(random_labels, 6)
# 训练生成网络
g_loss = adversarial_model.train_on_batch([z_noise2,
                                           sampled_labels], [1] * batch_size)
```

上述代码会使用一批次的输入训练生成网络。对抗模型的输入是 z_noise2 和 random_labels。

(16) 接着计算损失并输出。

```
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
print("d_loss:{}".format(d_loss))
print("g_loss:{}".format(g_loss))
# 将损失添加到相应列表中
gen_losses.append(g_loss)
dis_losses.append(d_loss)
```

(17) 然后将损失写入 TensorBoard，以进行可视化。

```
write_log(tensorboard, 'g_loss', np.mean(gen_losses), epoch)
write_log(tensorboard, 'd_loss', np.mean(dis_losses), epoch)
```

(18) 每训练 10 轮，采样并保存图像，如下所示。

```
if epoch % 10 == 0:
    images_batch = loaded_images[0:batch_size]
    images_batch = images_batch / 127.5 - 1.0
    images_batch = images_batch.astype(np.float32)

    y_batch = y[0:batch_size]
    z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))

    gen_images = generator.predict_on_batch([z_noise, y_batch])

    for i, img in enumerate(gen_images[:5]):
        save_rgb_img(img, path="results/img_{}_{}.png".format(epoch, i))
```

将该代码块放入轮循环中，每 10 轮生成一批次的假图像并写入结果目录。其中 `save_rgb_img()`是通用函数，其定义如下。

```
def save_rgb_img(img, path):
    """
    保存一张 rgb 图像
    """
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.imshow(img)
    ax.axis("off")
    ax.set_title("Image")

    plt.savefig(path)
    plt.close()
```

(19) 最后，添加如下代码来保存两个模型。

```
# 只保存权重
generator.save_weights("generator.h5")
discriminator.save_weights("discriminator.h5")

# 同时保存架构和权重
generator.save("generator.h5)
discriminator.save("discriminator.h5")
```

执行完本节中的代码，便完成了生成网络和判别网络的训练。完成上述步骤之后，生成网络会开始生成模糊的人脸图像。下面训练用于潜在向量初步近似的编码模型。

3.5.2 潜在向量初步近似

前面讲过，cGAN 并不能学习从图像到潜在向量的反向映射。而编码网络可以学习这种反向映射，并生成潜在向量，针对目标年龄生成人脸图像。下面训练编码网络。

前面定义了训练所需的超参数。下面训练编码网络，步骤如下。

(1) 首先构建编码网络。添加以下代码，构建并编译编码网络。

```
# 构建编码网络
encoder = build_encoder()
encoder.compile(loss=euclidean_distance_loss, optimizer='adam')
```

至此，还未定义 euclidean_distance_loss 函数。下面定义该函数，并在构建编码网络前添加它。

```
def euclidean_distance_loss(y_true, y_pred):
    """
    欧氏距离损失
    """
    return K.sqrt(K.sum(K.square(y_pred - y_true), axis=-1))
```

(2) 然后加载生成网络，如下所示。

```
generator.load_weights("generator.h5")
```

上一步成功训练并保存了生成网络的权重值，这里加载这些权重值。

(3) 接着采样一批次的潜在向量，如下所示。

```
z_i = np.random.normal(0, 1, size=(1000, z_shape))
```

(4) 然后采样一批次的随机年龄数字，并将其转换成独热编码向量，如下所示。

```
y = np.random.randint(low=0, high=6, size=(1000,), dtype=np.int64)
num_classes = len(set(y))
y = np.reshape(np.array(y), [len(y), 1])
y = to_categorical(y, num_classes=num_classes)
```

可以采样任意数量。这里采样了 1000 个值。

(5) 接着添加轮循环和批次步骤循环，如下所示。

```
for epoch in range(epochs):
    print("Epoch:", epoch)

    encoder_losses = []

    number_of_batches = int(z_i.shape[0] / batch_size)
    print("Number of batches:", number_of_batches)
    for index in range(number_of_batches):
        print("Batch:", index + 1)
```

(6) 从 1000 个样本中采样一批次的潜在向量和一批次的独热编码向量，如下所示。

```
z_batch = z_i[index * batch_size:(index + 1) * batch_size]
y_batch = y[index * batch_size:(index + 1) * batch_size]
```

(7) 接着使用预训练的生成网络生成假图像。

```
generated_images = generator.predict_on_batch([z_batch, y_batch])
```

(8) 然后使用生成网络生成的图像训练编码网络。

```
encoder_loss = encoder.train_on_batch(generated_images, z_batch)
```

(9) 每轮训练过后，将编码网络损失写入 TensorBoard 中，如下所示。

```
write_log(tensorboard, "encoder_loss", np.mean(encoder_losses), epoch)
```

(10) 最后，添加以下代码保存训练好的编码网络。

```
encoder.save_weights("encoder.h5")
```

至此，训练过的编码网络可以生成初始潜在向量了。下面介绍如何优化潜在向量近似。

3.5.3 潜在向量优化

前面成功训练了生成网络、判别网络以及编码网络。下面实现**人脸识别**（face recognition，FR）。对于给定的输入，人脸识别网络生成一个 128 维的嵌入。

操作步骤如下。

(1) 首先构建编码网络和生成网络，并加载其权重值。

```
encoder = build_encoder()
encoder.load_weights("encoder.h5")

# 加载生成网络
generator.load_weights("generator.h5")
```

(2) 然后创建一个网络，将图像形状从`(64, 64, 3)`转换成`(192, 192, 3)`，如下所示。

```
# 首先定义需要调用的所有函数
def build_image_resizer():
    input_layer = Input(shape=(64, 64, 3))

    resized_images = Lambda(lambda x: K.resize_images(
        x, height_factor=3, width_factor=3,
        data_format='channels_last'))(input_layer)

    model = Model(inputs=[input_layer], outputs=[resized_images])
    return model

image_resizer = build_image_resizer()
image_resizer.compile(loss=loss, optimizer='adam')
```

图像的高度和宽度需要大于 150 像素才能使用 FaceNet。可用上面的网络将图像转换成理想的格式。

(3) 构建人脸识别模型。

```
# 人脸识别模型
fr_model = build_fr_model(input_shape=fr_image_shape)
fr_model.compile(loss=loss, optimizer="adam")
```

有关 `build_fr_model()` 函数的更多信息，请访问：https://github.com/PacktPublishing/Generative-AdversarialNetworks-Projects/Age-cGAN/main.py。

(4) 接着创建另一个对抗模型。该模型含 3 个网络：编码网络、生成网络，以及人脸识别模型。

```
# 将人脸识别网络设置为“不可训练”
fr_model.trainable = False

# 输入层
input_image = Input(shape=(64, 64, 3))
input_label = Input(shape=(6,))

# 使用编码网络和生成网络
latent0 = encoder(input_image)
gen_images = generator([latent0, input_label])

# 将图像缩放至所需的形状
resized_images = Lambda(lambda x: K.resize_images(
    gen_images, height_factor=3, width_factor=3,
    data_format='channels_last'))(gen_images)
embeddings = fr_model(resized_images)

# 创建一个 Keras 模型，并确定网络的输入和输出
fr_adversarial_model = Model(inputs=[input_image, input_label],
                             outputs=[embeddings])

# 编译模型
fr_adversarial_model.compile(loss=euclidean_distance_loss,
                             optimizer=adversarial_optimizer)
```

(5) 添加轮循环，以及该循环下的批次步骤循环，如下所示。

```
for epoch in range(epochs):
    print("Epoch:", epoch)
    number_of_batches = int(len(loaded_images) / batch_size)
    print("Number of batches:", number_of_batches)
    for index in range(number_of_batches):
        print("Batch:", index + 1)
```

(6) 然后从真实图像的列表中采样一批次的图像。

```
# 采样并进行归一化处理
images_batch = loaded_images[index * batch_size:(index + 1) * batch_size]
images_batch = images_batch / 255.0
images_batch = images_batch.astype(np.float32)
# 采样一批次年龄的独热编码向量
y_batch = y[index * batch_size:(index + 1) * batch_size]
```

(7) 接着使用 FR 网络生成真实图像的嵌入。

```
images_batch_resized = image_resizer.predict_on_batch(images_batch)
real_embeddings = fr_model.predict_on_batch(images_batch_resized)
```

(8) 最后，训练该对抗模型，即训练编码网络和生成网络。

```
reconstruction_loss = fr_adversarial_model.train_on_batch([images_batch, y_batch],
                                                          real_embeddings)
```

(9) 将重构的损失写入 TensorBoard 中，以便进行可视化。

```
# 将重构损失写入Tensorboard 中
write_log(tensorboard, "reconstruction_loss",
          reconstruction_loss, index)
```

(10) 保存两个网络的权重。

```
# 保存两个网络的优化后的权重
generator.save_weights("generator_optimized.h5")
encoder.save_weights("encoder_optimized.h5")
```

祝贺！成功训练了用于实现人脸老化的 Age-cGAN。

3.5.4　损失可视化

启动 TensorBoard 服务器，将训练损失可视化。如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图，如图 3-3 所示。

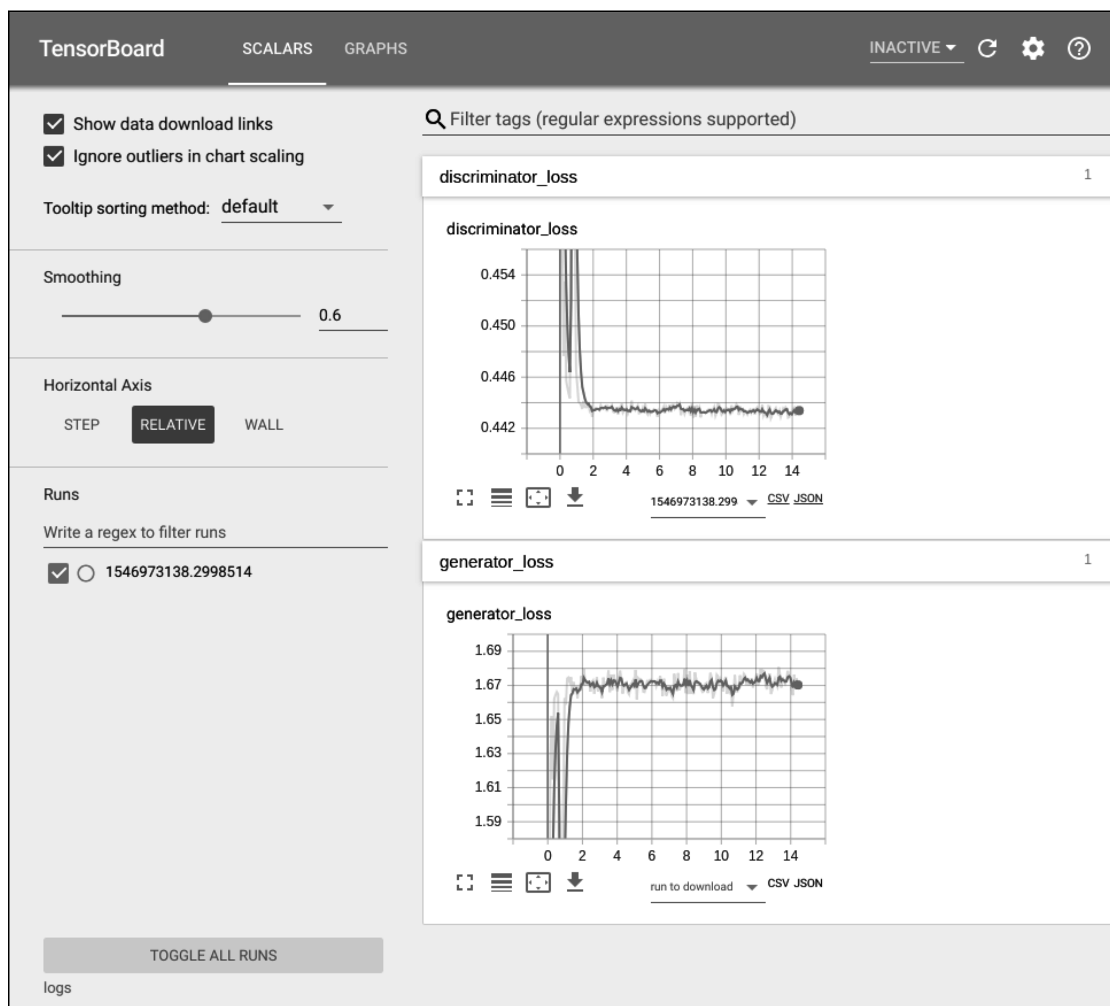


图 3-3 两种损失的曲线图

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

3.5.5 图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 3-4）。

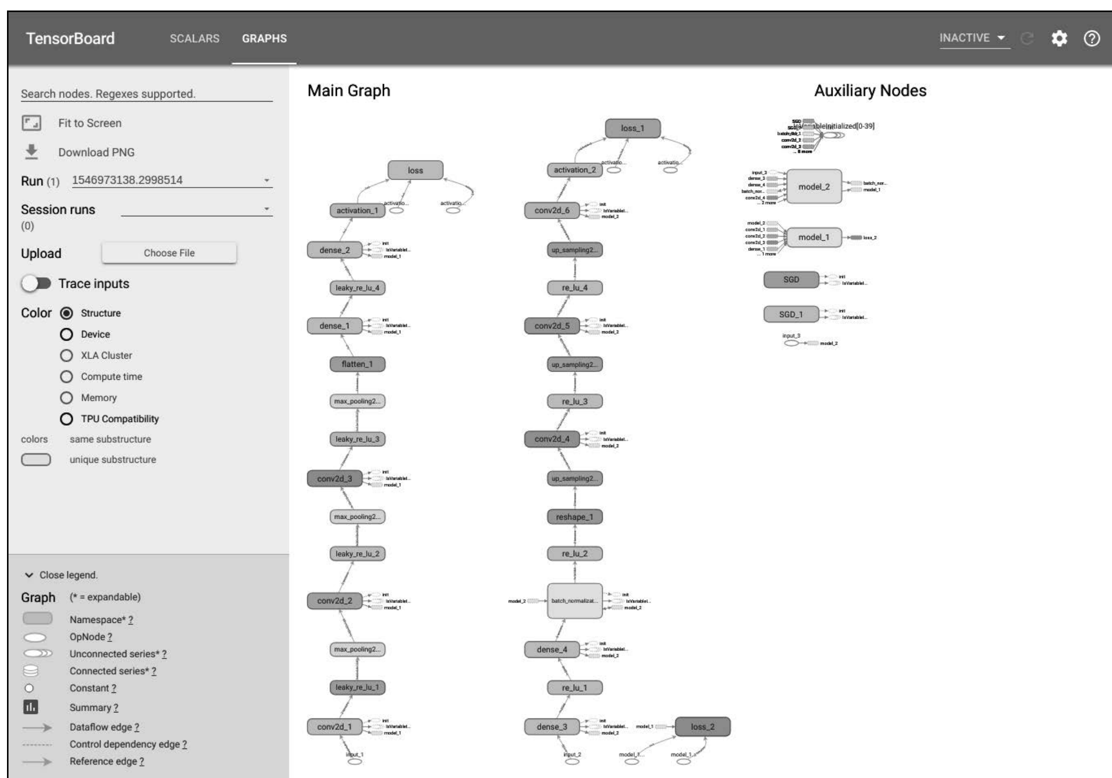


图 3-4　各图中的张量和不同运算的流

3.6　Age-cGAN 的实际应用

人脸老化在许多行业和消费者应用中都有广泛应用。

- **跨年龄人脸识别**。该技术可用于安全性应用，比如手机设备解锁或桌面电脑解锁。目前人脸识别系统存在无法随时间更新的问题。使用 Age-cGAN 的跨年龄人脸识别系统的生命周期会更长。
- **寻找失踪儿童**。这是 Age-cGAN 的一个有趣应用。儿童随着年龄的增长，其面部特征会发生变化，也就更难以辨识。Age-cGAN 可以模拟特定年龄的人脸。
- **娱乐**。例如某些手机应用可以呈现并分享某个朋友在特定年龄的照片。
- **电影中的面部特效**。人工模拟人物的老年面容既枯燥又耗时。Age-cGAN 可以加速该过程，并能降低创建和模拟人脸的成本。

3.7 小结

本章介绍了 Age-cGAN 及其架构，然后介绍了如何构建项目，以及 Age-cGAN 的 Keras 实现，接着使用来自维基百科的剪裁后的数据集训练了 Age-cGAN，并讲解了 Age-cGAN 的 3 个训练阶段。最后讨论了 Age-cGAN 的实际应用。

下一章将使用 GAN 的另一个变体 DCGAN 来生成动画人物。

第4章

使用 DCGAN 生成动画人物

众所周知，卷积层非常擅长处理图像。例如 Inception、AlexNet、视觉几何组（visual geometry group，VGG）和 ResNet 等神经网络的卷积层可以高效学习边缘、形状、复杂对象等很多重要特征。Ian Goodfellow 等人在论文“Generative Adversarial Nets”中提出了使用全连接层的 GAN。最初的 GAN 没有使用复杂的神经网络，比如 CNN、RNN 和 LSTM。DCGAN 的发展向使用 CNN 生成图像迈出了重要一步。DCGAN 使用卷积层替代全连接层。这是由 Alec Radford、Luke、Metz 和 Soumith Chintala 等研究者在论文“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”中提出的。此后 DCGAN 便广泛应用于各种图像生成任务中了。本章使用一种 DCGAN 架构生成动画人物。

本章将讨论以下主题。

- DCGAN 简介
- GAN 的架构细节
- 创建项目
- 准备训练数据集
- 用 Keras 实现可生成动画人物的 DCGAN
- 在动画人物数据集上训练 DCGAN
- 评估模型
- 通过超参数优化网络
- DCGAN 的实际应用

4.1 DCGAN 简介

无论是进行图像分类还是检测图像中的目标，CNN 在计算机视觉任务中表现不俗。研究者从 CNN 对图像的出色理解中获得启发，将其加入了 GAN 中。最初，官方 GAN 论文的作者只引入了包含全连接层的深度神经网络（DNN）。GAN 的原始实现中没有使用卷积层。早期 GAN 中生成网络和判别网络都使用全连接隐藏层。后来，一些论文作者提出 GAN 也可以使用不同的神经网络架构配置。

DCGAN 扩展了将卷积层加入判别网络和生成网络中的想法。DCGAN 的配置和普通 GAN 类似，由一个生成网络和一个判别网络组成。生成网络和判别网络都是包含卷积层的 DNN。训练 DCGAN 和训练普通 GAN 类似。第 1 章介绍了 GAN 中的各个网络参与一个非合作博弈。在该博弈中，判别网络将错误反向传播给生成网络，生成网络使用该错误来优化其权重。

稍后介绍这两个网络的架构。

DCGAN 的具体架构

前面提过，DCGAN 的两个网络都使用卷积层。重申一下，CNN 包含卷积层，其后是归一化层或池化层，之后是激活函数。在 DCGAN 中，判别网络接收图像，使用卷积层和池化层对其进行下采样，然后使用全连接分类层将图像分类为真的或假的。生成网络从潜在空间中获取随机噪声向量，然后通过上采样机制进行上采样，最终生成一张图像。隐藏层使用 Leaky ReLU 作为激活函数，并且使用系数介于 0.4 到 0.7 的随机失活来避免过拟合。

下面介绍两个网络的配置。

1. 配置生成网络

开始之前先看一下生成网络的架构（见图 4-1）。

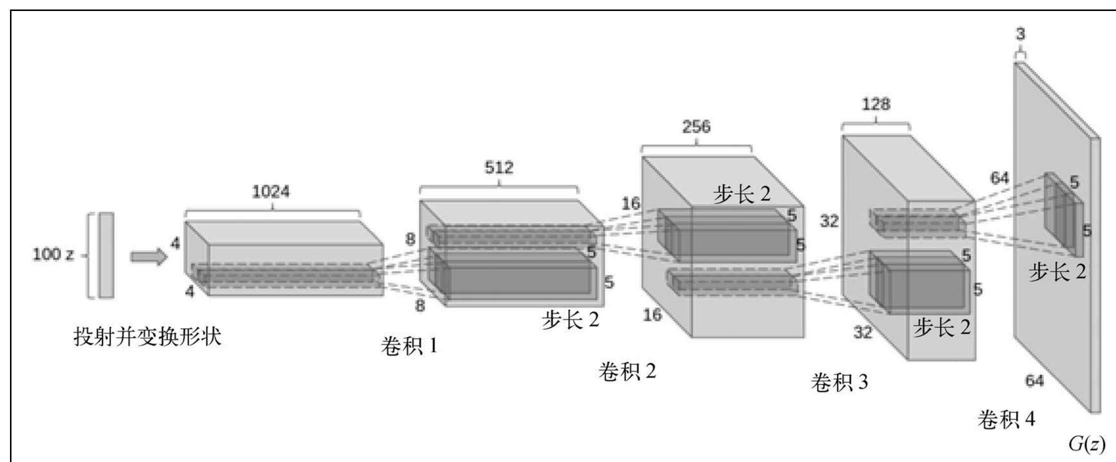


图 4-1　一个生成网络的架构

上图展示了生成网络架构的各层，以及生成网络生成一张分辨率为 64×64×3 的图像的过程。

DCGAN 的生成网络包含 10 层。它使用跨步卷积层来提高张量在空间中的分辨率。在 Keras 中，将上采样层和卷积层组合在一起等同于一个跨步卷积层。简单说来，生成网络接收从均匀概率分布中采样的噪声向量，然后不断对其进行变换，直到生成最终的图像。换言之，生成网络接

收形状为(batch_size, 100)的张量，输出形状为(batch_size, 64, 64, 3)的张量。

生成网络各层的配置如表 4-1 所示。

表 4-1

层序号	层名称	配置
1	输入层	input_shape=(batch_size, 100), output_shape=(batch_size, 100)
2	全连接层	neurons=2048,input_shape=(batch_size, 100), output_shape=(batch_size, 2048),activation='relu'
3	全连接层	neurons=16384,input_shape=(batch_size, 100), output_shape=(batch_size, 2048), batch_normalization=Yes,activation='relu'
4	形状变换层	input_shape=(batch_size=16384), output_shape=(batch_size, 8, 8, 256)
5	上采样层	size=(2, 2),input_shape=(batch_size, 8, 8, 256), output_shape=(batch_size, 16, 16, 256)
6	2D 卷积层	filters=128,kernel_size=(5, 5),strides=(1, 1), padding='same',input_shape=(batch_size, 16, 16, 256),output_shape=(batch_size, 16, 16, 128),activation='relu'
7	上采样层	size=(2, 2),input_shape=(batch_size, 16, 16, 128), output_shape=(batch_size, 32, 32, 128)
8	2D 卷积层	filters=64,kernel_size=(5, 5),strides=(1, 1), padding='same',activation=ReLU , input_shape=(batch_size, 32, 32, 128), output_shape=(batch_size, 32, 32, 64), activation='relu'
9	上采样层	size=(2, 2),input_shape=(batch_size, 32, 32, 64), output_shape=(batch_size, 64, 64, 64)
10	2D 卷积层	filters=3,kernel_size=(5, 5),strides=(1, 1), padding='same',activation=ReLU, input_shape=(batch_size, 64, 64, 64), output_shape=(batch_size, 64, 64, 3), activation='tanh'

下面讲解张量是如何从第一层流动到最后一层的。图 4-2 展示了不同层的输入和输出的形状。

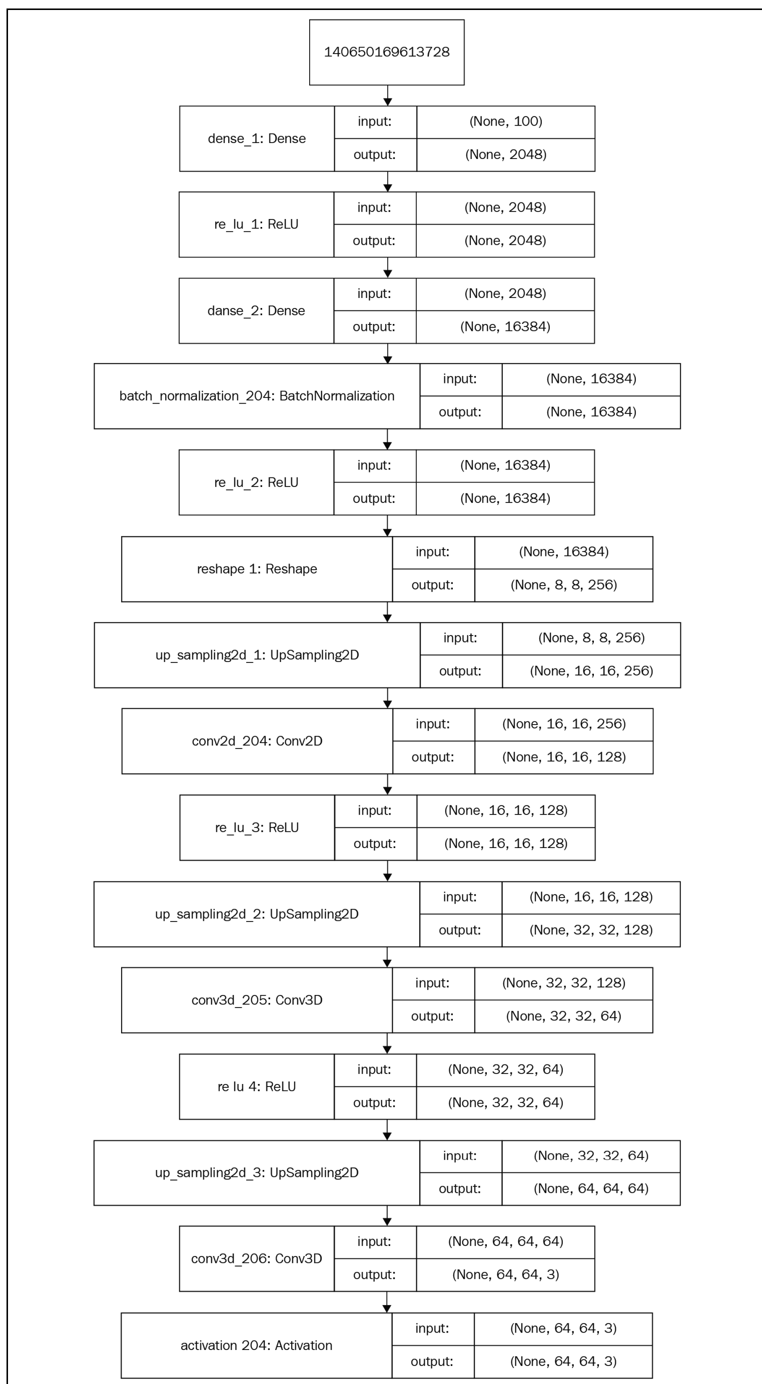


图 4-2　不同层的输入和输出的形状



为了保证该配置的有效性，需要使用 TensorFlow 后端以及 channels_last 格式。

2. 配置判别网络

进行下一步之前，先看一下判别网络的架构（见图 4-3）。

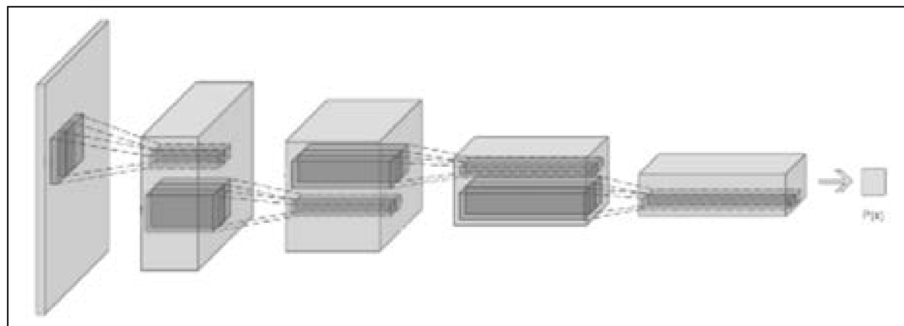


图 4-3 一个判别网络的架构

上图展示了判别网络架构的顶层概况。

前面提过，判别网络是包含 10 层（可以添加更多层）的 CNN。简单说来，它接收维度为 64×64×3 的图像，使用 2D 卷积层对其进行下采样，然后传递给全连接层进行分类。判别网络输出估测，判断给定图像是真是假。输出值为 0 或 1，输出 1 表示判别网络接收的图像为真，输出 0 表示该图像为假。

判别网络各层的配置如表 4-2 所示。

表 4-2

层 序 号	层 名 称	配 置
1	输入层	input_shape=(batch_size, 64, 64, 3), output_shape=(batch_size, 64, 64, 3)
2	2D 卷积层	filters=128,kernel_size=(5, 5),strides=(1, 1), padding='valid',input_shape=(batch_size, 64, 64, 3),output_shape=(batch_size, 64, 64, 128),activation='leakyrelu',leaky_relu_alpha=0.2
3	最大池化层	pool_size=(2, 2),input_shape=(batch_size, 64, 64, 128),output_shape=(batch_size, 32, 32, 128)
4	2D 卷积层	filters=256,kernel_size=(3, 3),strides=(1, 1), padding='valid',input_shape=(batch_size, 32, 32, 128),output_shape=(batch_size, 30, 30, 256),activation='leakyrelu',leaky_relu_alpha=0.2

（续）

层 序 号	层 名 称	配 置
5	最大池化层	pool_size=(2, 2), input_shape=(batch_size, 30, 30, 256), output_shape=(batch_size, 15, 15, 256)
6	2D 卷积层	filters=512,kernel_size=(3, 3),strides=(1, 1), padding='valid',input_shape=(batch_size, 15, 15, 256),output_shape=(batch_size, 13, 13, 512),activation='leakyrelu',leaky_relu_alpha=0.2
7	最大池化层	pool_size=(2, 2),input_shape=(batch_size, 13, 13, 512),output_shape=(batch_size, 6, 6, 512)
8	扁平化层	input_shape=(batch_size, 6, 6, 512), output_shape=(batch_size, 18432)
9	全连接层	neurons=1024,input_shape=(batch_size, 18432), output_shape=(batch_size, 1024), activation='leakyrelu','leakyrelu_alpha'=0.2
10	全连接层	neurons=1,input_shape=(batch_size, 1024), output_shape=(batch_size, 1),activation='sigmoid'

下面讲解张量是如何从第一层流动到最后一层的。图 4-4 展示了不同层的输入和输出的形状。

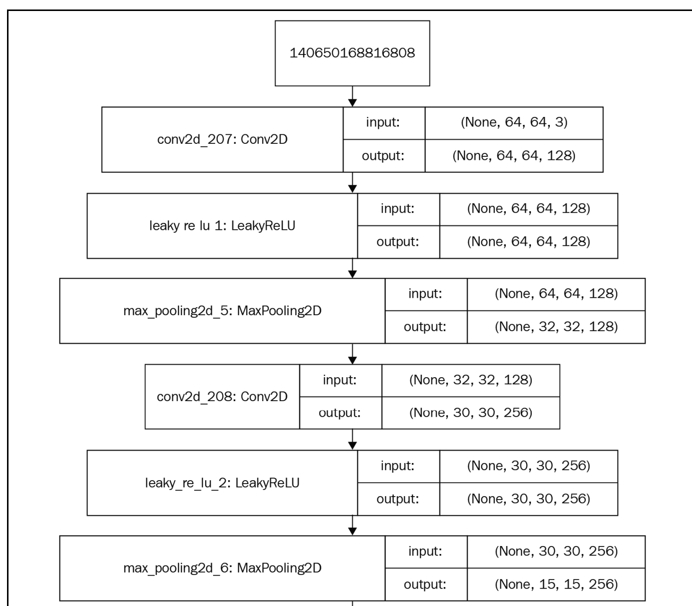


图 4-4 不同层的输入和输出的形状

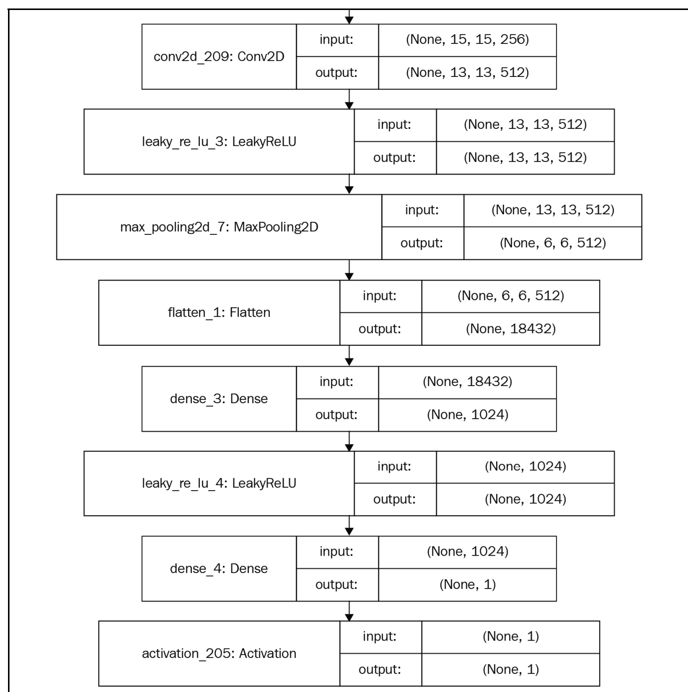


图 4-4 （续）



为了保证该配置的有效性，需要使用 TensorFlow 后端以及 channels_last 格式。

4.2 创建项目

前面已经克隆或下载了本书所有章节的完整代码。其中目录 Chapter04 包含本章的完整代码。执行如下命令创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter04。

```
cd Chapter04
```

(3) 然后为本项目创建 Python 虚拟环境。

```
virtualenv venv
virtualenv venv -p python3 # 创建一个使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建一个使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 接着启用新创建的虚拟环境。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 然后执行如下命令，安装 requirements.txt 文件中列出的所需程序。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

这样就成功创建了项目并安装了所需的依赖程序。下面处理数据集，包括下载和清洗数据集。

4.3　下载并准备动画人物数据集

训练 DCGAN 需要用到包含剪裁好的人物面部图像的动画人物数据集。收集该数据集有不同的方式，可以使用公开的数据集，也可以在不违反网站爬取规范的情况下自行爬取。本章爬取图像仅用于教学和演示。本章使用爬虫工具 gallery-dl 从 pixiv.net 上爬取图像。可用该命令行工具从 pixiv.net、exhentai.org、danbooru.donmai.us 等网站上下载图像集。

4.3.1　下载数据集

本节介绍安装依赖程序和下载数据集的各个步骤。启用为本项目创建的虚拟环境，然后执行以下命令。

(1) 执行如下命令下载 gallery-dl。

```
pip install --upgrade gallery-dl
```

(2) 也使用以下命令安装 gallery-dl 的最新开发版本。

```
pip install --upgrade
https://github.com/mikf/gallery-dl/archive/master.zip
```

(3) 如果以上命令不能正常运行，请按照官方软件仓库中的说明进行操作。

```
# gallery-dl 的官方 Github 仓库
https://github.com/mikf/gallery-dl
```

(4) 最后，执行以下命令，使用 gallery-dl 从 danbooru.donmai.us 上下载图像。

```
gallery-dl https://danbooru.donmai.us/posts?tags=face
```



下载图像，风险自担。上述信息仅用于教学，我们不支持非法爬取。这些图像是由所有者托管在平台上的，我们未获版权。如想商用，请联系网站所有者或所用内容的所有者。

4.3.2 探索数据集

在剪裁或缩放图像之前，先看一下所下载的图像（见图 4-5）。

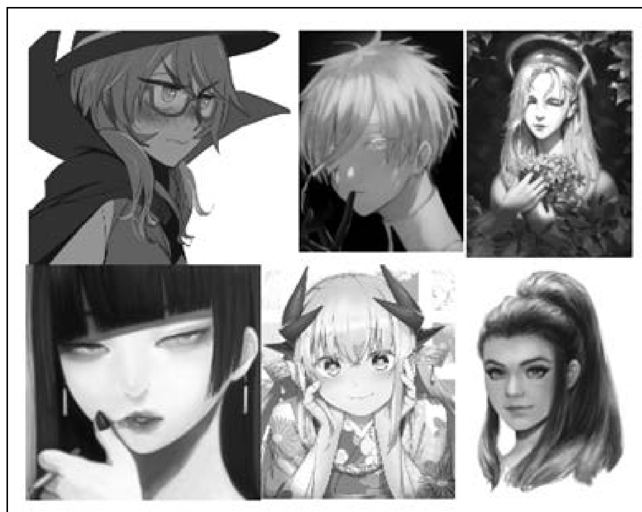


图 4-5 部分下载图像

如上图所示，有些图像包含了身体的其他部位，这些部位不应出现在训练图像中。下面从这些图像中剪裁出面部，并将图像缩放到训练所需的大小。

4.3.3 剪裁及缩放训练集图像

下面将图像中的面部剪裁出来。所使用的工具是 python-animeface，这是一个开源的 GitHub 仓库，可以通过命令行自动将图像中的面部剪裁出来。

剪裁和缩放图像的步骤如下。

(1) 首先下载 python-animeface。

```
pip install animeface
```

(2) 然后导入本任务所需模块。

```
import glob
import os
```

```
import animeface
from PIL import Image
```

(3) 接着定义参数。

```
total_num_faces = 0
```

(4) 然后对所有图像依次进行剪裁和缩放。

```
for index, filename in
enumerate(glob.glob('/path/to/directory/containing/images/*.*')):
```

(5) 在循环内部打开当前图像，并检测其中的面部。

```
try:
    # 打开图像
    im = Image.open(filename)

    # 检测面部
    faces = animeface.detect(im)
except Exception as e:
    print("Exception:{}".format(e))
    continue
```

(6) 接着获取图像中检测到的面部的坐标。

```
fp = faces[0].face.pos

# 获取图像中检测到的面部的坐标
coordinates = (fp.x, fp.y, fp.x+fp.width, fp.y+fp.height)
```

(7) 从图像中剪裁出面部。

```
# 裁剪图像
cropped_image = im.crop(coordinates)
```

(8) 对剪裁出来的面部图像进行缩放，使其维度为`(64, 64)`。

```
# 缩放图像
cropped_image = cropped_image.resize((64, 64), Image.ANTIALIAS)
```

(9) 最后，将剪裁和缩放后的图像保存到目标目录。

```
cropped_image.save("/path/to/directory/to/store/cropped/images/filename.png"))
```

将完整代码包装成 Python 函数，如下所示。

```
import glob
import os

import animeface
from PIL import Image
```

```
total_num_faces = 0

for index, filename in
enumerate(glob.glob('/path/to/directory/containing/images/*.*')):
    # 打开图像并检测面部
    try:
        im = Image.open(filename)
        faces = animeface.detect(im)
    except Exception as e:
        print("Exception:{}".format(e))
        continue

    # 如果在当前图像中没有找到面部
    if len(faces) == 0:
        print("No faces found in the image")
        continue

    fp = faces[0].face.pos

    # 获取图像中检测到的面部的坐标
    coordinates = (fp.x, fp.y, fp.x+fp.width, fp.y+fp.height)

    # 剪裁图像
    cropped_image = im.crop(coordinates)

    # 缩放图像
    cropped_image = cropped_image.resize((64, 64), Image.ANTIALIAS)

    # 呈现经过剪裁和缩放的图像
    # cropped_image.show()

    # 将图像保存到输出目录中
cropped_image.save("/path/to/directory/to/store/cropped/images/filename.png"))

    print("Cropped image saved successfully")
    total_num_faces += 1
    print("Number of faces detected till now:{}".format(total_num_faces))

print("Total number of faces:{}".format(total_num_faces))
```

上述脚本会加载已下载的所有图像，使用 python-animeface 库检测面部，然后从原始图像中将面部剪裁出来，接着会将剪裁出来的图像缩放到 64×64 大小。如果想改变图像维度，可以相应地调整生成网络和判别网络的架构。准备就绪后，下面着手实现网络。

4.4 使用 Keras 实现 DCGAN

本节使用 Keras 框架实现 DCGAN。Keras 是一个元框架，使用 TensorFlow 或者 Theano 作为后端。Keras 提供了操作神经网络的高级别 API。相比于 TensorFlow 等低级别框架，Keras 拥有一些预构建的神经网络层、优化器、正则项、初始化器以及数据预处理层，可进行简单的原型构建。下面编写生成网络的实现代码。

4.4.1　生成网络

前面提过，生成网络包含一些 2D 卷积层、一些上采样层、一个形状变换层，以及一个批归一化层。在 Keras 中，层可以实现任何运算，甚至激活函数也是层，可以像普通的全连接层一样添加到模型中。

实现生成网络的步骤如下。

(1) 首先创建一个 Keras 的 Sequential 模型。

```
gen_model = Sequential()
```

(2) 然后添加一个具有 2048 个节点的全连接层，以及一个激活函数 tanh。

```
gen_model.add(Dense(units=2048))
gen_model.add(Activation('tanh'))
```

(3) 接着添加第二层，一个具有 16 384 个神经元的全连接层。然后添加一个批归一化层，使用默认超参数，设置激活函数为 tanh。

```
gen_model.add(Dense(256*8*8))
gen_model.add(BatchNormalization())
gen_model.add(Activation('tanh'))
```

第二个全连接层的输出是形状为`(16384,)`的张量。该全连接层的神经元数量相当于一个形状为`(256, 8, 8)`的张量。

(4) 再添加一个形状变换层，将上一层的张量形状变换为`(batch_size, 8, 8, 256)`。

```
# 形状变换层
gen_model.add(Reshape((8, 8, 256), input_shape=(256*8*8,)))
```

(5) 接着添加一个 2D 上采样层，将形状从`(8, 8, 256)`变换为`(16, 16, 256)`。上采样大小为`(2, 2)`，即把张量的大小增加到原来的两倍。至此，已有 256 个形状为 16×16 的张量。

```
gen_model.add(UpSampling2D(size=(2, 2)))
```

(6) 然后添加一个 2D 卷积层。该层会使用特定数量的过滤器对张量进行 2D 卷积。此处使用 64 个过滤器，以及一个形状为`(5, 5)`的卷积核。

```
gen_model.add(Conv2D(128, (5, 5), padding='same'))
gen_model.add(Activation('tanh'))
```

(7) 接着添加一个 2D 上采样层，将张量的形状从`(batch_size, 16, 16, 64)`变换为`(batch_size, 32, 32, 64)`。

```
gen_model.add(UpSampling2D(size=(2, 2)))
```

一个 2D 上采样层将张量的行和列分别重复 0 次和 1 次。

(8) 然后添加第 2 个 2D 卷积层，使用 64 个过滤器，以及一个大小为(5, 5)的卷积核，设置激活函数为 tanh。

```
gen_model.add(Conv2D(64, (5, 5), padding='same'))
gen_model.add(Activation('tanh'))
```

(9) 接着添加一个 2D 上采样层，将张量的形状从(batch_size, 32, 32 , 64)变换为(batch_size, 64, 64, 64)。

```
gen_model.add(UpSampling2D(size=(2, 2)))
```

(10) 最后，添加第 3 个 2D 卷积层，使用 3 个过滤器，以及一个大小为(5, 5)的卷积核，设置激活函数为 tanh。

```
gen_model.add(Conv2D(3, (5, 5), padding='same'))
gen_model.add(Activation('tanh'))
```

生成网络会输出形状为(batch_size, 64, 64, 3)的张量。这些张量中的单个图像张量类似于维度为 64×64 的图像，具有红、绿、蓝（RGB）3 个通道。

将生成网络的完整代码包装成 Python 函数，如下所示。

```
def get_generator():
    gen_model = Sequential()

    gen_model.add(Dense(input_dim=100, output_dim=2048))
    gen_model.add(LeakyReLU(alpha=0.2))

    gen_model.add(Dense(256 * 8 * 8))
    gen_model.add(BatchNormalization())
    gen_model.add(LeakyReLU(alpha=0.2))

    gen_model.add(Reshape((8, 8, 256), input_shape=(256 * 8 * 8,)))
    gen_model.add(UpSampling2D(size=(2, 2)))

    gen_model.add(Conv2D(128, (5, 5), padding='same'))
    gen_model.add(LeakyReLU(alpha=0.2))

    gen_model.add(UpSampling2D(size=(2, 2)))

    gen_model.add(Conv2D(64, (5, 5), padding='same'))
    gen_model.add(LeakyReLU(alpha=0.2))

    gen_model.add(UpSampling2D(size=(2, 2)))

    gen_model.add(Conv2D(3, (5, 5), padding='same'))
    gen_model.add(LeakyReLU(alpha=0.2))
    return gen_model
```

这样就创建好了生成网络。下面创建判别网络。

4.4.2　判别网络

前面提过，判别网络包含 3 个 2D 卷积层，每个后面都跟着一个激活函数，然后是两个最大池化层。判别网络的末端是两个全连接层，用作分类层。首先简单介绍一下判别网络中的各层。

- 所有卷积层都使用 LeakyReLU 作为激活函数，其 alpha 值设置为 0.2。
- 卷积层分别包含 128 个、256 个和 512 个过滤器，其卷积核大小分别为(5, 5)、(3, 3)和(3, 3)。
- 卷积层之后是一个扁平化层，将输入变换为一维张量。
- 扁平化层之后是两个全连接层，分别有 1024 个神经元和 1 个神经元。
- 第一个全连接层使用 LeakyReLU 作为激活函数，第二个使用 sigmoid 作为激活函数进行二分类。稍后训练判别网络判别图像的真假。

实现判别网络的步骤如下。

(1) 首先创建一个 Keras 的 Sequential 模型。

```
dis_model = Sequential()
```

(2) 添加一个 2D 卷积层，接收形状为(64, 64, 3)的输入图像。该层的超参数如下所示。此外，添加一个 alpha 值为 0.2 的 LeakyReLU 作为激活函数。

- 过滤器数量：128
- 卷积核大小：(5, 5)
- 填充方式：same

```
dis_model.add(Conv2D(filters=128, kernel_size=5, padding='same',
              input_shape=(64, 64, 3)))
dis_model.add(LeakyReLU(alpha=0.2))
```

(3) 接着添加一个池大小为(2, 2)的 2D 最大池化层，以对图像表示进行下采样，通过对图像的非重叠子区域使用最大过滤器来实现。

```
dis_model.add(MaxPooling2D(pool_size=(2, 2)))
```

第一层输出张量的形状为(batch_size, 32, 32, 128)。

(4) 接着添加另一个 2D 卷积层，配置如下。

- 过滤器数量：256
- 卷积核大小：(3, 3)
- 激活函数：LeakyReLU，alpha 值为 0.2
- 2D 最大池化层的池大小：(2, 2)

```
dis_model.add(Conv2D(filters=256, kernel_size=3))
dis_model.add(LeakyReLU(alpha=0.2))
dis_model.add(MaxPooling2D(pool_size=(2, 2)))
```

该层输出张量的形状为(batch_size, 30, 30, 256)。

(5) 然后添加第三个 2D 卷积层，配置如下。

- **过滤器数量**：512
- **卷积核大小**：(3, 3)
- **激活函数**：LeakyReLU，alpha 值为 0.2
- **2D 最大池化层的池大小**：(2, 2)

```
dis_model.add(Conv2D(512, (3, 3)))
dis_model.add(LeakyReLU(alpha=0.2))
dis_model.add(MaxPooling2D(pool_size=(2, 2)))
```

该层输出张量的形状为(batch_size, 13, 13, 256)。

(6) 接着添加一个扁平化层，将输入扁平化，但不影响批大小。该层输出一个二维张量。

```
dis_model.add(Flatten())
```

该扁平化层输出张量的形状为(batch_size, 18432,)。

(7) 然后添加一个具有 1024 个神经元的全连接层，使用 alpha 值为 0.2 的 LeakyReLU 作为激活函数。

```
dis_model.add(Dense(1024))
dis_model.add(LeakyReLU(alpha=0.2))
```

(8) 最后，添加一个神经元数量为 1 的全连接层用于进行二分类。sigmoid 函数是进行二分类的最佳选择，它提供了类别概率。

```
dis_model.add(Dense(1))
dis_model.add(Activation('sigmoid'))
```

判别网络生成形状为(batch_size, 1)的输出张量，包含类别的概率。

将生成网络的完整代码包装成 Python 函数，如下所示。

```
def get_discriminator():
    dis_model = Sequential()
    dis_model.add(
        Conv2D(128, (5, 5),
               padding='same',
               input_shape=(64, 64, 3))
    )
    dis_model.add(LeakyReLU(alpha=0.2))
    dis_model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
    dis_model.add(Conv2D(256, (3, 3)))
    dis_model.add(LeakyReLU(alpha=0.2))
    dis_model.add(MaxPooling2D(pool_size=(2, 2)))

    dis_model.add(Conv2D(512, (3, 3)))
    dis_model.add(LeakyReLU(alpha=0.2))
    dis_model.add(MaxPooling2D(pool_size=(2, 2)))

    dis_model.add(Flatten())
    dis_model.add(Dense(1024))

    dis_model.add(LeakyReLU(alpha=0.2))

    dis_model.add(Dense(1))
    dis_model.add(Activation('sigmoid'))

    return dis_model
```

这样就成功实现了判别网络和生成网络。下面在 4.2 节准备的数据集上训练模型。

4.5　训练 DCGAN

训练 DCGAN 类似于训练普通 GAN，过程分为 4 步。

(1) 加载数据集。
(2) 构建并编译两个网络。
(3) 训练判别网络。
(4) 训练生成网络。

下面详述各步骤。

首先定义变量和超参数。

```
dataset_dir = "/Path/to/dataset/directory/*.*"
batch_size = 128
z_shape = 100
epochs = 10000
dis_learning_rate = 0.0005
gen_learning_rate = 0.0005
dis_momentum = 0.9
gen_momentum = 0.9
dis_nesterov = True
gen_nesterov = True
```

确定了训练用的不同超参数后，下面介绍如何加载训练数据集。

4.5.1 加载样本

训练 DCGAN 需要将数据集加载到内存，并定义按批次加载到内存的机制。加载数据集步骤如下。

(1) 首先加载之前经过剪裁、缩放，并保存在 cropped 文件夹中的全部图像。正确指定目录路径，以便 `glob.glob` 函数创建一个包含所有文件的列表。使用 scipy.mics 模块中的 imread 函数读取图像。加载目录中全部图像的代码如下。

```
# 加载图像
all_images = []
for index, filename in
enumerate(glob.glob('/Path/to/cropped/images/directory/*.*')):
    image = imread(filename, flatten=False, mode='RGB')
    all_images.append(image)
```

(2) 接着创建包含全部图像的 ndarray，最终形状会是`(total_num_images, 64, 64, 3)`。然后将所有图像归一化。

```
# 转换为 NumPy ndarray 格式
X = np.array(all_images)
X = (X - 127.5) / 127.5
```

加载数据集后，下面介绍如何构建并编译网络。

4.5.2 构建并编译网络

下面构建并编译训练所需的两个网络。

(1) 首先定义训练所需的优化器，如下所示。

```
# 定义优化器
dis_optimizer = SGD(lr=dis_learning_rate, momentum=dis_momentum,
                    nesterov=dis_nesterov)
gen_optimizer = SGD(lr=gen_learning_rate, momentum=gen_momentum,
                    nesterov=gen_nesterov)
```

(2) 然后创建生成网络模型的一个实例，编译生成网络模型（编译时会初始化权重参数、优化器算法、损失函数，以及其他核心步骤）。

```
gen_model = build_generator()
gen_model.compile(loss='binary_crossentropy',
                  optimizer=gen_optimizer)
```

使用 `binary_crossentropy` 作为生成网络的损失函数、`gen_optimizer` 作为其优化器。

(3) 接着创建判别网络模型的一个实例并编译，如下所示。

```
dis_model = build_discriminator()
dis_model.compile(loss='binary_crossentropy',
                  optimizer=dis_optimizer)
```

类似地，使用 binary_crossentropy 作为判别网络的损失函数、dis_optimizer 作为优化器。

(4) 然后创建对抗模型，即把两个网络包含在一个模型中。对抗模型的架构如下所示。

输入→生成网络→判别网络→输出

创建并编译对抗模型的代码如下。

```
adversarial_model = Sequential()
adversarial_model.add(gen_model)
dis_model.trainable = False
adversarial_model.add(dis_model)
```

训练该网络时不训练判别网络，所以在将判别网络添加到对抗模型之前将其设置为“不可训练”。

编译对抗模型，如下所示。

```
adversarial_model.compile(loss='binary_crossentropy',
                          optimizer=gen_optimizer)
```

使用 binary_crossentropy 作为对抗网络的损失函数、gen_optimizer 作为其优化器。

在开始训练之前添加 TensorBoard，对损失进行可视化。如下所示。

```
tensorboard = TensorBoard(log_dir="logs/{}".format(time.time()),
                          write_images=True, write_grads=True, write_graph=True)
tensorboard.set_model(gen_model)
tensorboard.set_model(dis_model)
```

下面训练网络多次迭代，需要创建一个循环，指定运行轮数。每轮会在一个大小为 128 的小批量上训练网络。下面计算需要处理的批次数量。

```
for epoch in range(epcohs):
    print("Epoch is", epoch)
    number_of_batches = int(X.shape[0] / batch_size)
    print("Number of batches", number_of_batches)
    for index in range(number_of_batches):
```

下面具体介绍训练过程。首先解释 DCGAN 训练所涉及的几个步骤。

- 两个网络的权重最初都是随机设定的。
- 训练 DCGAN 的标准流程是，首先在一批次样本上训练判别网络。

- 该步骤要用到假样本和真样本。真样本已经有了，这里需要生成假样本。
- 生成假样本需要在均匀概率分布中构建形状为`(100,)`的潜在向量，然后将其传递给未训练过的生成网络。生成网络会生成假样本，用于训练判别网络。
- 将真图像和假图像组合成新的样本图像集。还需要创建一个标签数组：标签 1 代表真图像，标签 0 代表假图像。

4.5.3 训练判别网络

训练判别网络的步骤如下。

(1) 首先从正态分布中采样一批次噪声向量，如下所示。

```
z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))
```

使用 NumPy 库的 np.random 模块的 `normal()`函数进行采样。

(2) 然后从所有图像的集合中采样一批次真实图像。

```
image_batch = X[index * batch_size:(index + 1) * batch_size]
```

(3) 接着使用生成网络生成一批次假图像。

```
generated_images = gen_model.predict_on_batch(z_noise)
```

(4) 然后创建真标签和假标签。

```
y_real = np.ones(batch_size) - np.random.random_sample(batch_size) * 0.2
y_fake = np.random.random_sample(batch_size) * 0.2
```

(5) 接着在真图像和真标签上训练判别网络。

```
dis_loss_real = dis_model.train_on_batch(image_batch, y_real)
```

(6) 类似地，在假图像和假标签上训练判别网络。

```
dis_loss_fake = dis_model.train_on_batch(generated_images, y_fake)
```

(7) 然后计算平均损失，并输出到控制台。

```
d_loss = (dis_loss_real+dis_loss_fake)/2
print("d_loss:", d_loss)
```

前面一直在训练判别网络，下面训练生成网络。

4.5.4 训练生成网络

需要通过训练对抗模型来训练生成网络。训练对抗模型时，需要锁定判别网络，只训练生成网络。前面训练过了判别网络，因此不再训练。训练对抗模型的步骤如下。

(1) 首先再创建一批次噪声向量。从正态分布中采样这些噪声向量。

```
z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))
```

(2) 然后在这批次噪声向量上训练对抗模型，如下所示。

```
g_loss = adversarial_model.train_on_batch(z_noise, [1] * batch_size)
```

使用一批次噪声向量和真实标签训练对抗模型。其中，**真实标签**是所有值都为 1 的向量。同时这是在训练生成网络欺骗判别网络。该过程是通过向判别网络提供所有值都为 1 的向量实现的。这一步生成网络将会收到判别网络的反馈，并据此优化自身。

(3) 最后，将生成网络的损失输出到控制台，以追踪损失。

```
print("g_loss:", g_loss)
```

可以使用一种被动方法评估训练过程。每训练 10 轮生成一批假图像，并人工检查这些图像的质量。

```
if epoch % 10 == 0:
    z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))
    gen_images1 = gen_model.predict_on_batch(z_noise)

    for img in gen_images1[:2]:
        save_rgb_img(img, "results/one_{}.png".format(epoch))
```

可以根据这些图像判定是否继续训练。如果生成的高分辨率图像质量很好，可以停止训练；否则就继续训练，直到模型够好。

这样就在动画人物数据集上成功训练了一个 DCGAN。下面使用该模型生成动画人物的图像。

4.5.5　生成图像

生成图像需要从潜在空间采样噪声向量。可用 NumPy 的 `uniform()` 函数从均匀概率分布中生成向量。生成图像的步骤如下。

(1) 通过添加下面这行代码，创建一个维度为 `(batch_size, 100)` 的噪声向量。

```
z_noise = np.random.normal(0, 1, size=(batch_size, z_shape))
```

(2) 然后，使用生成网络的 `predict_on_batch` 方法生成图像。该方法接收上一步创建的噪声向量。

```
gen_images = gen_model.predict_on_batch(z_noise)
```

(3) 图像已经生成，通过添加下面这行代码来保存它。创建一个名为 results 的目录，存储生成的图像。

```
imsave('results/image_{}.jpg'.format(epoch),gen_images[0])
```

至此，可以打开这些生成的图像，检验所得模型的质量了。这是一种估计模型性能的被动方法。

4.5.6　保存模型

在 Keras 中，保存模型只需一行代码，如下所示。

```
# 指定生成网络模型的路径
gen_model.save("directory/for/the/generator/model.h5")
```

类似地，保存判别网络模型的代码如下。

```
# 指定判别网络模型的路径
dis_model.save("directory/for/the/discriminator/model.h5")
```

需要通过训练对抗模型来训练生成网络。训练对抗模型时，会锁定判别网络，只训练生成网络。前面训练过判别网络了，因此不再训练。训练对抗模型的步骤如下。

4.5.7　生成图像可视化

100 轮训练后，生成网络会开始生成不错的图像。下面看一下这些生成的图像。

100 轮训练后，图像如图 4-6 所示。



图 4-6　100 轮训练后生成的图像

200 轮训练后，图像如图 4-7 所示。



图 4-7　200 轮训练后生成的图像

通常网络训练 10 000 轮后可生成非常不错的图像。

4.5.8　损失可视化

启动 TensorBoard 服务器，将训练损失可视化。如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图，如图 4-8 所示。

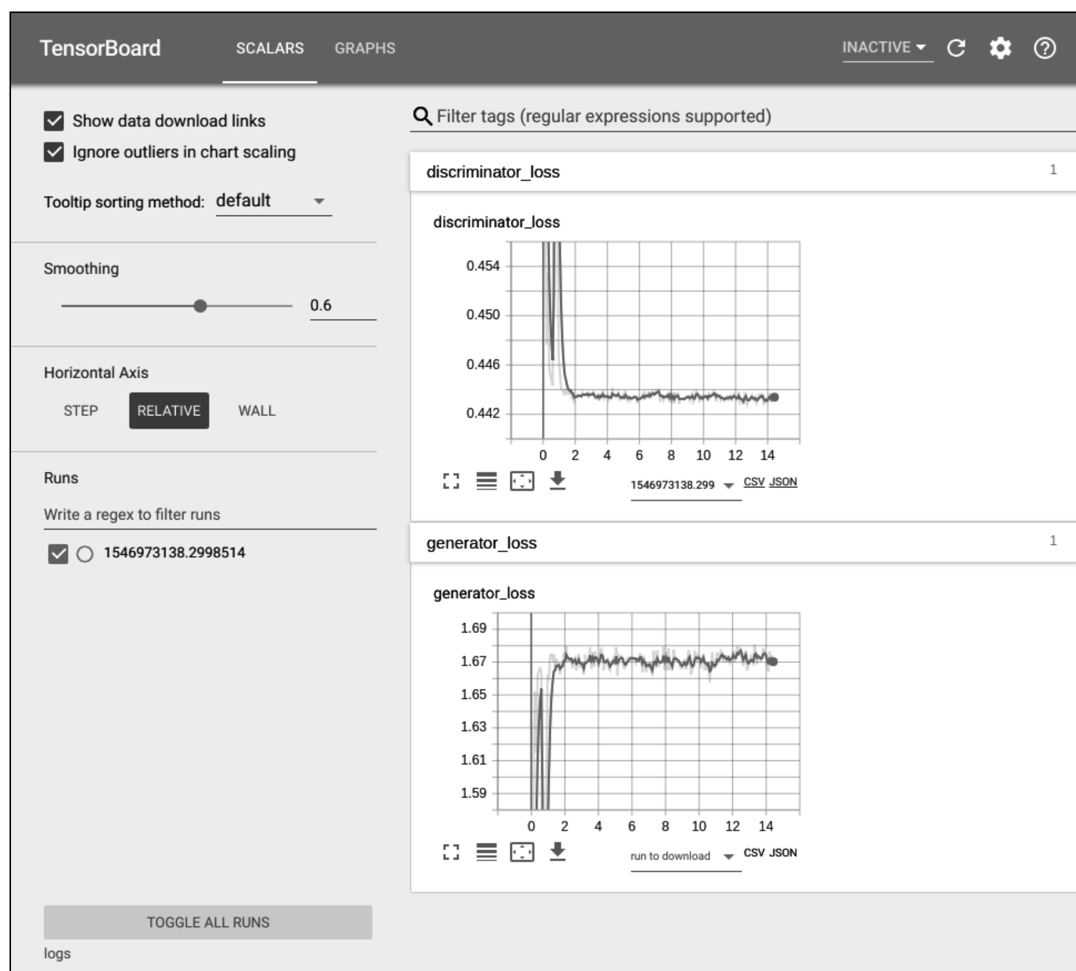


图 4-8　两种损失的曲线图

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

4.5.9 图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 4-9）。

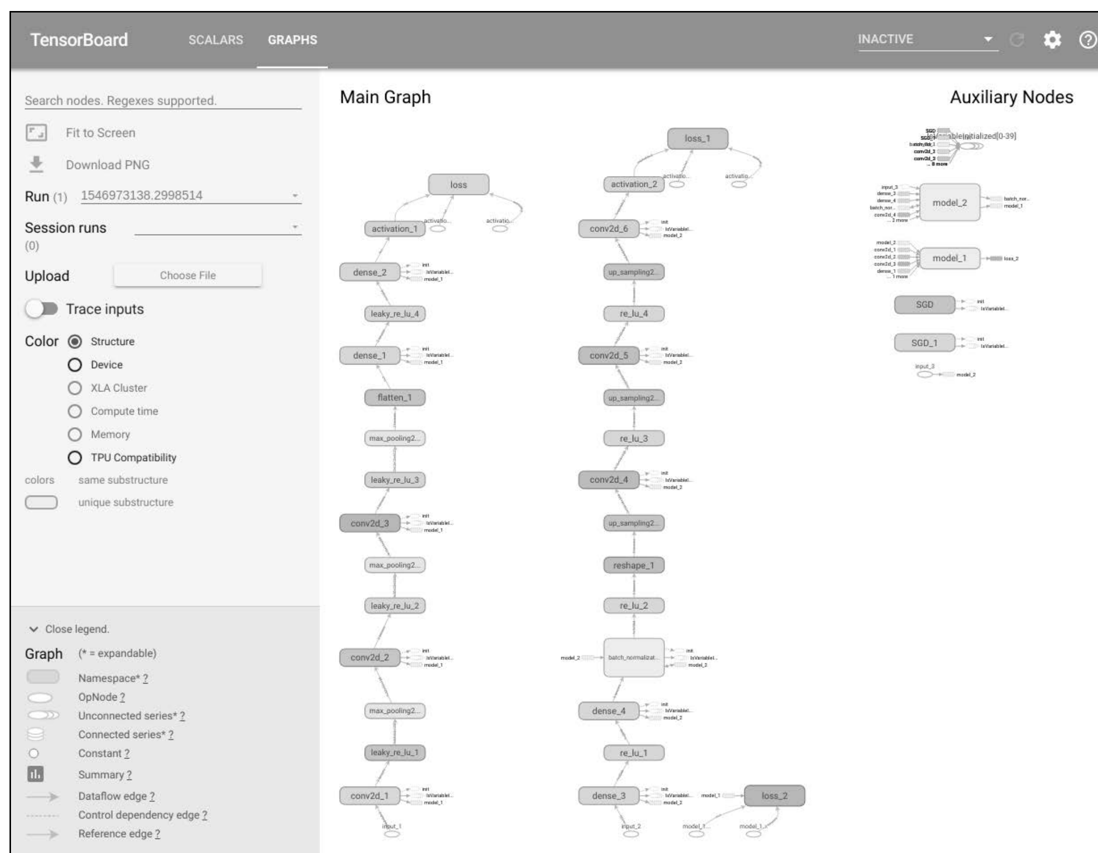


图 4-9 各图中的张量和不同运算的流

4.5.10 超参数调优

超参数是模型的属性，在训练中固定不变。参数不同，准确性也可能不同。下面介绍一些常用的超参数。

- ❑ 学习速率
- ❑ 批大小
- ❑ 训练轮数
- ❑ 生成网络的优化器

- 判别网络的优化器
- 层数
- 全连接层的节点数量
- 激活函数
- 损失函数

4.4 节的学习速率是固定的：生成网络模型和判别网络模型都使用 0.0005。批大小是 128。调节这些数值可优化模型。如果模型无法生成较好的图像，可以尝试更改这些数值，然后重新运行模型。

4.6 DCGAN 的实际应用

DCGAN 可以针对不同的用例进行定制。DCGAN 的部分实际应用如下。

- **生成动画人物**。目前，动画制作者需要使用计算机软件人工绘制人物，有时需要在纸上绘制。这种人工处理的过程往往非常耗时。使用 DCGAN 可以很快生成新的动画人物，从而加速创作过程。
- **增强数据集**。训练有监督机器学习模型时，为了确保模型质量，需要使用很大的数据集。DCGAN 可以增强已有数据集，增加有监督模型训练所需数据集的大小。
- **生成 MNIST 字符**。MNIST 数据集包含 60 000 个手写数字的图像。如果要训练复杂的有监督学习模型，MNIST 数据集是不够大的。DCGAN 训练完成后可以生成新数字，扩充原始数据集。
- **生成人脸图像**。DCGAN 使用卷积神经网络，擅长生成逼真的图像。
- **提取特征**。训练完成后，可以从判别网络的中间层中提取特征。这些特征对于风格迁移和人脸识别等任务都非常实用。在风格迁移任务中，需要生成图像的内部表示，用于计算风格和内容的损失。关于风格迁移的更多信息，请参考论文“A Neural Algorithm of Artistic Style”。

4.7 小结

本章介绍了深度卷积生成对抗网络。首先简单介绍了 DCGAN，然后详细介绍了 DCGAN 架构，之后创建了项目并安装了所需的依赖程序，接着介绍了如何下载和准备数据集，随后用 Keras 实现了网络，并在数据集上进行了训练。训练完成之后，使用模型生成了新的动画人物。本章还讨论了 DCGAN 的实际应用。

下一章将介绍 SRGAN，它用于生成高分辨率图像。

第 5 章

使用 SRGAN 生成逼真图像

SRGAN（super-resolution generative adversarial network，超分辨率生成对抗网络）可以使用低像素图像生成具有更多细节、质量更高的超分辨率图像。最初，人们使用 CNN 来获得高分辨率图像，其模型训练快且准确度高。然而在某些情况下，CNN 无法修复更为精细的细节，往往生成模糊的图像。本章使用 Keras 框架实现一个能生成高分辨率图像的 SRGAN。SRGAN 最初是由 Christian Ledig、Lucas Theis、Ferenc Huszar、Jose Caballero 和 Andrew Cunningham 等人在论文“Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network”中提出的。

本章将讨论以下主题。

- SRGAN 简介
- 创建项目
- 下载 CelebA 数据集
- SRGAN 的 Keras 实现
- 训练 SRGAN，以及优化网络
- SRGAN 的实际应用

5.1 SRGAN 简介

和其他 GAN 类似，SRGAN 包括一个生成网络和一个判别网络，两个都是深度神经网络。它们的功能如下。

- **生成网络**：接收维度为 64×64×3 的低分辨率图像，经过卷积层和上采样层的一系列处理，生成一个形状为 256×256×3 的超分辨率图像
- **判别网络**：接收高分辨率图像，试图判断给定图像是真（属于真实数据样本）还是假（由生成网络生成的）

5.1.1 SRGAN 架构

SRGAN 中的两个网络都是深度卷积神经网络，包含卷积层和上采样层。每个卷积层后面是

一个批归一化运算和一个激活层。稍后详述两个网络的细节。图 5-1 展示了 SRGAN 的架构。

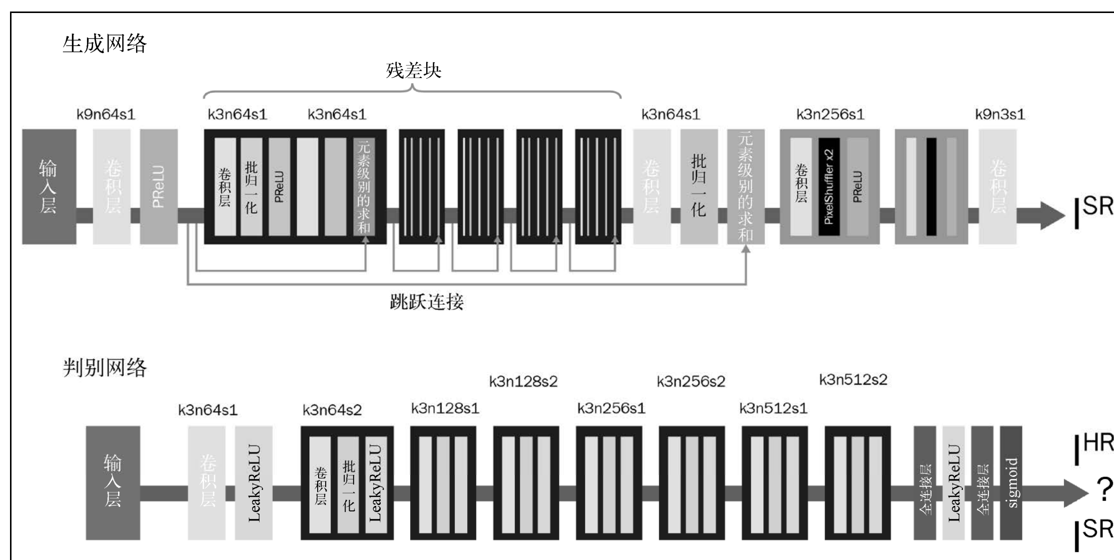


图 5-1　生成网络和判别网络的架构，图中标明了每个卷积层相应的卷积核大小（k），特征映射的数量（n）以及步长（s）

下面详细介绍两个网络的架构。

1. 生成网络的架构

前面提过，生成网络是深度卷积神经网络，由下面这些块组成。

- 前残差块
- 残差块
- 后残差块
- 上采样块
- 最终的卷积层

下面逐一探讨。

- **前残差块**：包含一个 2D 卷积层，使用 ReLU 作为激活函数。配置如下（见表 5-1）。

表　5-1

层　名　称	超　参　数	输入形状	输出形状
2D 卷积层	`Filters=64, kernel_size=3, strides=1, padding='same', activation='relu'`	(64, 64, 3)	(64, 64, 64)

- **残差块**：包含两个 2D 卷积层。每个卷积层后面都有一个批归一化层，其 `Momentum` 值为 0.8。配置如下（见表 5-2）。

表 5-2

层 名 称	超 参 数	输入形状	输出形状
2D 卷积层	Filters=64, kernel_size=3, strides=1, padding='same', activation='relu'	(64, 64, 64)	(64, 64, 64)
批归一化层	Momentum=0.8	(64, 64, 64)	(64, 64, 64)
2D 卷积层	Filters=64, kernel_size=3, strides=1, padding='same'	(64, 64, 64)	(64, 64, 64)
批归一化层	Momentum=0.8	(64, 64, 64)	(64, 64, 64)
加法层	无	(64, 64, 64)	(64, 64, 64)

加法层计算该块的输入张量和最后的批归一化层的输出之和。生成网络包含 16 个配置如上的残差块。

- **后残差块**：后残差块也包含一个 2D 卷积层，使用 ReLU 作为激活函数。卷积层后面是一个批归一化层，其 `Momentum` 值为 0.8。后残差块的配置如下（见表 5-3）。

表 5-3

层 名 称	超 参 数	输入形状	输出形状
2D 卷积层	Filters=64, kernel_size=3, strides=1, padding='same'	(64, 64, 64)	(64, 64, 64)
批归一化层	Momentum=0.8	(64, 64, 64)	(64, 64, 64)

- **上采样块**：上采样块包含一个上采样层和一个 2D 卷积层，使用 ReLU 作为激活函数。生成网络有两个上采样块。第一个上采样块的配置如下（见表 5-4）。

表 5-4

层 名 称	超 参 数	输入形状	输出形状
2D 上采样层	Size=(2, 2)	(64, 64, 64)	(128, 128, 64)
2D 卷积层	Filters=256, kernel_size=3, strides=1, padding='same', activation='relu'	(128, 128, 256	(128, 128, 256)

5

第二个上采样块的配置如下（见表5-5）。

表　5-5

层　名　称	超　参　数	输入形状	输出形状
2D上采样层	Size=(2, 2)	(128, 128, 256)	256, 256, 256)
2D卷积层	Filters=256, kernel_size=3, strides=1, padding='same', activation='relu'	256, 256, 256)	256, 256, 256)

❑ **最后的卷积层**：最后一层是一个2D卷积层，使用tanh作为激活函数。该层生成一个形状为(256, 256, 3)的图像。最后一层的配置如下（见表5-6）。

表　5-6

层　名　称	超　参　数	输入形状	输出形状
2D卷积层	Filters=3, kernel_size=9, strides=1, padding='same', activation='tanh'	(256, 256, 256)	(256, 256, 3)



这些超参数是针对Keras设计的。如果使用其他框架，请做相应调整。

2. 判别网络的架构

判别网络也是深度卷积网络。它包含8个卷积块和两个全连接层。每个卷积块后面都有一个批归一化层。网络末端是两个全连接层，相当于一个分类块。最后一层估测给定图像属于真数据集或假数据集的概率。判别网络的具体配置如表5-7所示。

表　5-7

层　名　称	超　参　数	输入形状	输出形状
输入层	无	(256, 256, 3)	(256, 256, 3)
2D卷积层	filters=64, kernel_size=3, strides=1, padding='same', activation='leakyrelu'	(256, 256, 3)	(256, 256, 64)
2D卷积层	filters=64, kernel_size=3, strides=2, padding='same', activation='leakyrelu'	(256, 256, 64)	(128, 128, 64)
批归一化层	momentum=0.8	(128, 128, 64)	(128, 128, 64)
2D卷积层	filters=128, kernel_size=3, strides=1, padding='same', activation='leakyrelu'	(128, 128, 64)	(128, 128, 128)
批归一化层	momentum=0.8	(128, 128, 128)	(128, 128, 128)

（续）

层 名 称	超 参 数	输入形状	输出形状
2D 卷积层	filters=128, kernel_size=3, strides=2, padding='same', activation='leakyrelu'	(128, 128, 128)	(64, 64, 128)
批归一化层	momentum=0.8	(64, 64, 128)	(64, 64, 128)
2D 卷积层	filters=256, kernel_size=3, strides=1, padding='same', activation='leakyrelu'	(64, 64, 128)	(64, 64, 256)
批归一化层	momentum=0.8	(64, 64, 256)	(64, 64, 256)
2D 卷积层	filters=256, kernel_size=3, strides=2, padding='same', activation='leakyrelu'	(64, 64, 256)	(32, 32, 256)

介绍过了两个网络，下面介绍训练 SRGAN 所需的目标函数。

5.1.2 训练目标函数

为了训练 SRGAN，需要将目标函数（也称“损失函数”）最小化。SRGAN 的目标函数称为**感知损失**（perceptual loss）函数，是两个损失函数的加权和。这两个损失函数如下所示。

- 内容损失
- 对抗损失

下面详细介绍内容损失和对抗损失。

1. 内容损失

内容损失有两类，分别是：

- 像素级 MSE 损失
- VGG 损失

下面具体讨论这些损失。

- **像素级 MSE 损失**

内容损失是真实图像的每个像素和生成图像的每个像素之间的均方误差。像素级 MSE 损失用于计算生成图像和真实图像之间的差异大小。方法如下。

$$l_{\mathrm{MSE}}^{SB}=\frac{1}{r^2WH}\sum_{x=1}^{rW}\sum_{y=1}^{rH}(I_{x,y}^{HR}-G_{\theta_G}(I^{LR})x,y)^2$$

其中，$G_{\theta_G}(I^{LR})$ 表示生成网络生成的高分辨率图像，I_{HR} 表示从真实数据集中采样的高分辨率图像。

- **VGG 损失**

VGG 损失也是内容损失函数，对生成图像和真实图像进行计算。VGG19 是流行的深度神经网络，主要用于图像分类。VGG19 是由 Simonyan 和 Zisserman 在论文“Very Deep Convolutional Networks for Large-Scale Image Recognition”中提出的。经过预训练的 VGG19 网络的中间层可以用作特征提取器，从生成图像和真实图像中提取特征映射。VGG 损失基于这些提取的特征映射，计算生成图像和真实图像的特征映射之间的欧氏距离。计算公式如下。

$$l_{\mathrm{VGG}/i,j}^{SR}=\frac{1}{W_{i,j}H_{i,j}}\sum_{x=1}^{W_{i,j}}\sum_{y=1}^{H_{i,j}}(\Phi_{i,j}(I^{HR})x,y-\Phi_{i,j}(G_{\theta_G}(I^{LR}))x,y)^2$$

其中，$\Phi_{i,j}$ 表示 VGG19 网络生成的特征映射，$\Phi_{i,j}(I^{HR})$ 表示从真实图像中提取的特征映射，$\Phi_{i,j}(G_{\theta_G}(I^{LR}))$ 表示从生成的高分辨率图像中提取的特征映射。该公式计算的是生成图像和真实图像的特征映射之间的欧氏距离。

这两种内容损失都可以用于训练 SRGAN。本章使用 VGG 损失来实现。

2. 对抗损失

对抗损失是根据判别网络返回的概率计算的。在对抗模型中，判别网络接收生成网络生成的图像。计算对抗损失的公式如下。

$$l_{\mathrm{Gen}}^{SR}=\sum_{n=1}^{N}-\log D_{\theta_G}(G_{\theta_G}(I^{LR}))$$

其中，$G_{\theta_G}(I^{LR})$ 表示生成的图像，$D_{\theta_G}(G_{\theta_G}(I^{LR}))$ 表示生成的图像为真的概率。

感知损失函数是内容损失和对抗损失的加权和，公式如下。

$$l^{SR}=1.0*l_X^{SR}+0.001*l_{\mathrm{Gen}}^{SR}$$

其中，l^{SR} 表示感知损失总和。l_X^{SR} 是内容损失，可以是像素级 MSE 损失或 VGG 损失。

通过最小化感知损失值，生成网络试图骗过判别网络。随着感知损失值降低，生成网络开始生成更为真实的图像。

下面开始构建项目。

5.2 创建项目

前面已经克隆或下载了本书所有章节的完整代码。其中目录 Chapter05 包含本章的完整代码。执行如下命令以创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter05。

```
cd Chapter05
```

(3) 然后为本项目创建一个 Python 虚拟环境。

```
virtualenv venv
virtualenv venv -p python3 # 创建一个使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建一个使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 接着启用新创建的虚拟环境。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 然后执行以下命令，安装 requirements.txt 文件中列出的所需的库。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

这样就创建好了项目，并安装了所需的依赖程序。下面处理数据集，并介绍如何下载数据集和变换格式。

5.3　下载 CelebA 数据集

本章使用大型的 CelebA（CelebFaces Attributes）数据集。该数据集包含 202 599 张名人面部图像。



该数据集仅用于非商业性研究，不得商用。如想商用，需获图片所有者授权。

本章使用 CelebA 数据集训练 SRGAN。下载数据集及提取图像的步骤如下。

(1) 下载该数据集。
(2) 执行以下命令，从下载的 img_align_celeba.zip 文件中提取图像。

```
unzip img_align_celeba.zip
```

下载数据集并提取图像后，下面使用 Keras 来实现 SRGAN。

5.4　SRGAN 的 Keras 实现

前面提过，SRGAN 由 3 个神经网络构成，分别是 1 个生成网络、1 个判别网络，以及 1 个在 Imagenet 数据集上预训练过的 VGG19 网络。下面编写这些网络的实现代码。首先实现生成网络。

在开始编写代码之前，首先创建一个 Python 文件 main.py，并导入核心模块，如下所示。

```
import glob
import os

import numpy as np
import tensorflow as tf
from keras import Input
from keras.applications import VGG19
from keras.callbacks import TensorBoard
from keras.layers import BatchNormalization, Activation, LeakyReLU, Add, \
    Dense, PReLU, Flatten
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.models import Model
from keras.optimizers import Adam
from keras_preprocessing.image import img_to_array, load_img
from scipy.misc import imsave
```

5.4.1　生成网络

前面介绍过生成网络的架构。下面用 Keras 框架编写生成网络的各层，然后使用 Keras 框架的函数式 API 创建一个 Keras 模型。

在 Keras 中实现生成网络的步骤如下。

(1) 首先定义生成网络所需的超参数。

```
residual_blocks = 16
momentum = 0.8
input_shape = (64, 64, 3)
```

(2) 然后创建一个输入层，为网络提供输入，如下所示。

```
input_layer = Input(shape=input_shape)
```



输入层接收形状为`(64, 64, 3)`的输入图像，然后传递给网络的下一层。

(3) 接着添加前残差块（2D 卷积层），配置如下。

- ❑ 过滤器数量：64
- ❑ 卷积核大小：9

- 步长：1
- 填充方式：same
- 激活函数：ReLU

```
gen1 = Conv2D(filters=64, kernel_size=9, strides=1, padding='same',
              activation='relu')(input_layer)
```

(4) 然后为残差块编写函数，如下所示。

```
def residual_block(x):
    """
    残差块
    """
    filters = [64, 64]
    kernel_size = 3
    strides = 1
    padding = "same"
    momentum = 0.8
    activation = "relu"

    res = Conv2D(filters=filters[0], kernel_size=kernel_size,
                 strides=strides, padding=padding)(x)
    res = Activation(activation=activation)(res)
    res = BatchNormalization(momentum=momentum)(res)

    res = Conv2D(filters=filters[1], kernel_size=kernel_size,
                 strides=strides, padding=padding)(res)
    res = BatchNormalization(momentum=momentum)(res)

    # 添加 res 和 x
    res = Add()([res, x])
    return res
```

(5) 接着使用上一步定义的 `residual_block` 函数，添加 16 个残差块。

```
res = residual_block(gen1)
for i in range(residual_blocks - 1):
    res = residual_block(res)
```

前残差块的输出会传递给第一个残差块。第一个残差块的输出传递给第二个残差块，以此类推，直到第 16 个残差块。

(6) 然后添加后残差块（一个 2D 卷积层，然后是一个批归一化层），配置如下。

- 过滤器数量：64
- 卷积核大小：3
- 步长：1
- 填充方式：same
- 是否使用批归一化：是（`momentum=0.8`）

5

```
gen2 = Conv2D(filters=64, kernel_size=3, strides=1, padding='same')(res)
gen2 = BatchNormalization(momentum=momentum)(gen2)
```

(7) 接着添加一个 Add 层，取前残差块的输出 gen1 和后残差块的输出 gen2 之和。这一层输出一个形状类似的张量。

```
gen3 = Add()([gen2, gen1])
```

(8) 然后添加一个上采样块，配置如下。

- 上采样大小：2
- 过滤器数量：256
- 卷积核大小：3
- 步长：1
- 填充方式：same
- 激活函数：PReLU

```
gen4 = UpSampling2D(size=2)(gen3)
gen4 = Conv2D(filters=256, kernel_size=3, strides=1, padding='same')(gen4)
gen4 = Activation('relu')(gen4)
```

(9) 再添加一个上采样块，配置如下。

- 上采样大小：2
- 过滤器数量：256
- 卷积核大小：3
- 步长：1
- 填充方式：same
- 激活函数：PReLU

```
gen5 = UpSampling2D(size=2)(gen4)
gen5 = Conv2D(filters=256, kernel_size=3, strides=1, padding='same')(gen5)
gen5 = Activation('relu')(gen5)
```

(10) 最后，添加输出卷积层，配置如下。

- 过滤器数量：3（和通道数量相等）
- 卷积核大小：9
- 步长：1
- 填充方式：same
- 激活函数：tanh

```
gen6 = Conv2D(filters=3, kernel_size=9, strides=1, padding='same')(gen5)
output = Activation('tanh')(gen6)
```

定义好了生成网络里的各层，就可以创建 Keras 模型了。前面使用 Keras 的函数式 API 定义了一个 Keras 的顺序图，下面为网络指定输入和输出，创建一个 Keras 模型。

(11) 创建一个 Keras 模型，指定模型的输入和输出，如下所示。

```
model = Model(inputs=[input_layer], outputs=[output], name='generator')
```

这样就创建好了生成网络的 Keras 模型。将生成网络的完整代码包装成 Python 函数，如下所示。

```
def build_generator():
    """
    使用下面定义的超参数值创建一个生成网络
    返回：生成网络模型
    """
    residual_blocks = 16
    momentum = 0.8
    input_shape = (64, 64, 3)

    # 生成网络的输入层
    input_layer = Input(shape=input_shape)

    # 添加前残差块
    gen1 = Conv2D(filters=64, kernel_size=9, strides=1, padding='same',
                  activation='relu')(input_layer)

    # 添加 16 个残差块
    res = residual_block(gen1)
    for i in range(residual_blocks - 1):
    res = residual_block(res)

    # 添加后残差块
    gen2 = Conv2D(filters=64, kernel_size=3, strides=1, padding='same')(res)
    gen2 = BatchNormalization(momentum=momentum)(gen2)

    # 取前残差块 (gen1) 的输出和后残差块 (gen2) 的输出之和
    gen3 = Add()([gen2, gen1])

    # 添加上采样块
    gen4 = UpSampling2D(size=2)(gen3)
    gen4 = Conv2D(filters=256, kernel_size=3, strides=1, padding='same')(gen4)
    gen4 = Activation('relu')(gen4)

    # 再添加一个上采样块
    gen5 = UpSampling2D(size=2)(gen4)
    gen5 = Conv2D(filters=256, kernel_size=3, strides=1, padding='same')(gen5)
    gen5 = Activation('relu')(gen5)

    # 输出卷积层
    gen6 = Conv2D(filters=3, kernel_size=9, strides=1, padding='same')(gen5)
    output = Activation('tanh')(gen6)
```

```
    # Keras 模型
    model = Model(inputs=[input_layer], outputs=[output],
                  name='generator')
    return model
```

生成网络的 Keras 模型创建后，下面创建判别网络的 Keras 模型。

5.4.2　判别网络

前面介绍过判别网络的架构。下面使用 Keras 框架编写判别网络各层，然后使用 Keras 框架的函数式 API 创建一个 Keras 模型。

在 Keras 中实现判别网络的步骤如下。

(1) 首先定义判别网络所需的超参数。

```
leakyrelu_alpha = 0.2
momentum = 0.8
input_shape = (256, 256, 3)
```

(2) 然后添加一个输入层，为网络提供输入，如下所示。

```
input_layer = Input(shape=input_shape)
```

(3) 接着添加一个卷积块，配置如下。

- 过滤器数量：64
- 卷积核大小：3
- 步长：1
- 填充方式：same
- 激活函数：LeakyReLU，alpha 值为 0.2

```
dis1 = Conv2D(filters=64, kernel_size=3, strides=1, padding='same')(input_layer)
dis1 = LeakyReLU(alpha=leakyrelu_alpha)(dis1)
```

(4) 然后再添加 7 个卷积块，配置如下。

- 过滤器数量：64，128，128，256，256，512，512
- 卷积核大小：3，3，3，3，3，3，3
- 步长：2，1，2，1，2，1，2
- 填充方式：same（每个卷积层都是）
- 激活函数：LeakyReLU，其 alpha 值为 0.2（每个卷积层都是）

```
# 添加第 2 个卷积块
dis2 = Conv2D(filters=64, kernel_size=3, strides=2, padding='same')(dis1)
dis2 = LeakyReLU(alpha=leakyrelu_alpha)(dis2)
dis2 = BatchNormalization(momentum=momentum)(dis2)
```

```
# 添加第 3 个卷积块
dis3 = Conv2D(filters=128, kernel_size=3, strides=1, padding='same')(dis2)
dis3 = LeakyReLU(alpha=leakyrelu_alpha)(dis3)
dis3 = BatchNormalization(momentum=momentum)(dis3)

# 添加第 4 个卷积块
dis4 = Conv2D(filters=128, kernel_size=3, strides=2, padding='same')(dis3)
dis4 = LeakyReLU(alpha=leakyrelu_alpha)(dis4)
dis4 = BatchNormalization(momentum=0.8)(dis4)

# 添加第 5 个卷积块
dis5 = Conv2D(256, kernel_size=3, strides=1, padding='same')(dis4)
dis5 = LeakyReLU(alpha=leakyrelu_alpha)(dis5)
dis5 = BatchNormalization(momentum=momentum)(dis5)

# 添加第 6 个卷积块
dis6 = Conv2D(filters=256, kernel_size=3, strides=2, padding='same')(dis5)
dis6 = LeakyReLU(alpha=leakyrelu_alpha)(dis6)
dis6 = BatchNormalization(momentum=momentum)(dis6)

# 添加第 7 个卷积块
dis7 = Conv2D(filters=512, kernel_size=3, strides=1, padding='same')(dis6)
dis7 = LeakyReLU(alpha=leakyrelu_alpha)(dis7)
dis7 = BatchNormalization(momentum=momentum)(dis7)

# 添加第 8 个卷积块
dis8 = Conv2D(filters=512, kernel_size=3, strides=2, padding='same')(dis7)
dis8 = LeakyReLU(alpha=leakyrelu_alpha)(dis8)
dis8 = BatchNormalization(momentum=momentum)(dis8)
```

5

(5) 接着添加一个具有 1024 个节点的全连接层，配置如下。

- 节点：1024
- 激活函数：LeakyReLU，alph 值为 0.2

```
dis9 = Dense(units=1024)(dis8)
dis9 = LeakyReLU(alpha=0.2)(dis9)
```

(6) 然后添加一个返回概率的全连接层，如下所示。

```
output = Dense(units=1, activation='sigmoid')(dis9)
```

(7) 最后，创建一个 Keras 模型，指明网络的输入和输出。

```
model = Model(inputs=[input_layer], outputs=[output],
              name='discriminator')
```

将判别网络的完整代码包装成函数，如下所示。

```
def build_discriminator():
    """
    使用下面定义的超参数值创建一个判别网络
```

```
返回：判别网络模型
"""
leakyrelu_alpha = 0.2
momentum = 0.8
input_shape = (256, 256, 3)

input_layer = Input(shape=input_shape)

# 添加第 1 个卷积块
dis1 = Conv2D(filters=64, kernel_size=3, strides=1, padding='same')(input_layer)
dis1 = LeakyReLU(alpha=leakyrelu_alpha)(dis1)

# 添加第 2 个卷积块
dis2 = Conv2D(filters=64, kernel_size=3, strides=2, padding='same')(dis1)
dis2 = LeakyReLU(alpha=leakyrelu_alpha)(dis2)
dis2 = BatchNormalization(momentum=momentum)(dis2)

# 添加第 3 个卷积块
dis3 = Conv2D(filters=128, kernel_size=3, strides=1, padding='same')(dis2)
dis3 = LeakyReLU(alpha=leakyrelu_alpha)(dis3)
dis3 = BatchNormalization(momentum=momentum)(dis3)

# 添加第 4 个卷积块
dis4 = Conv2D(filters=128, kernel_size=3, strides=2, padding='same')(dis3)
dis4 = LeakyReLU(alpha=leakyrelu_alpha)(dis4)
dis4 = BatchNormalization(momentum=0.8)(dis4)

# 添加第 5 个卷积块
dis5 = Conv2D(256, kernel_size=3, strides=1, padding='same')(dis4)
dis5 = LeakyReLU(alpha=leakyrelu_alpha)(dis5)
dis5 = BatchNormalization(momentum=momentum)(dis5)

# 添加第 6 个卷积块
dis6 = Conv2D(filters=256, kernel_size=3, strides=2, padding='same')(dis5)
dis6 = LeakyReLU(alpha=leakyrelu_alpha)(dis6)
dis6 = BatchNormalization(momentum=momentum)(dis6)

# 添加第 7 个卷积块
dis7 = Conv2D(filters=512, kernel_size=3, strides=1, padding='same')(dis6)
dis7 = LeakyReLU(alpha=leakyrelu_alpha)(dis7)
dis7 = BatchNormalization(momentum=momentum)(dis7)

# 添加第 8 个卷积块
dis8 = Conv2D(filters=512, kernel_size=3, strides=2, padding='same')(dis7)
dis8 = LeakyReLU(alpha=leakyrelu_alpha)(dis8)
dis8 = BatchNormalization(momentum=momentum)(dis8)

# 添加一个全连接层
dis9 = Dense(units=1024)(dis8)
dis9 = LeakyReLU(alpha=0.2)(dis9)

# 最后一个全连接层，用于进行分类
output = Dense(units=1, activation='sigmoid')(dis9)
```

```
    model = Model(inputs=[input_layer], outputs=[output], name='discriminator')
    return model
```

这样就创建好了判别网络的 Keras 模型。下面构建 5.1 节讲过的 VGG19 网络。

5.4.3 VGG19 网络

本章使用预训练的 VGG19 网络从生成的图像和真实图像中提取特征映射。下面使用 Keras 的预训练权重来构建并编译 VGG19 网络。

(1) 首先确定输入的形状。

```
input_shape = (256, 256, 3)
```

(2) 然后加载预训练的 VGG19，并指明模型的输出。

```
vgg = VGG19(weights="imagenet")
vgg.outputs = [vgg.layers[9].output]
```

(3) 接着创建一个符号式 `input_tensor`，用作 VGG19 网络的符号式输入，如下所示。

```
input_layer = Input(shape=input_shape)
```

(4) 然后使用 VGG19 提取特征。

```
features = vgg(input_layer)
```

(5) 最后，创建一个 Keras 模型，并指明网络的输入和输出。

```
model = Model(inputs=[input_layer], outputs=[features])
```

将 VGG19 模型的完整代码包装成函数，如下所示。

```
def build_vgg():
    """
    构建 VGG 网络来提取图像特征
    """
    input_shape = (256, 256, 3)

    # 加载在 Imagenet 数据集上预训练过的 VGG19 模型
    vgg = VGG19(weights="imagenet")
    vgg.outputs = [vgg.layers[9].output]

    input_layer = Input(shape=input_shape)

    # 提取特征
    features = vgg(input_layer)

    # 创建 Keras 模型
    model = Model(inputs=[input_layer], outputs=[features])
    return model
```

5.4.4 对抗网络

对抗网络是由生成网络、判别网络和 VGG19 结合而成的。下面创建一个对抗网络。

创建对抗网络的步骤如下。

(1) 首先为网络创建一个输入层。

```
input_low_resolution = Input(shape=(64, 64, 3))
```

对抗网络接收形状为`(64, 64, 3)`的图像，因此需要创建输入层。

(2) 然后使用生成网络生成假的高分辨率图像，如下所示。

```
fake_hr_images = generator(input_low_resolution)
```

(3) 接着使用 VGG19 网络从假图像中提取特征，如下所示。

```
fake_features = vgg(fake_hr_images)
```

(4) 然后将对抗网络中的判别网络设置为“不可训练”。

```
discriminator.trainable = False
```

训练生成网络时不训练判别网络，因此将判别网络设置为“不可训练”。

(5) 接着将假图像传递给判别网络。

```
output = discriminator(fake_hr_images)
```

(6) 最后，创建一个 Keras 模型，即对抗网络。

```
model = Model(inputs=[input_low_resolution], outputs=[output, fake_features])
```

(7) 将对抗模型的完整代码包装成 Python 函数。

```
def build_adversarial_model(generator, discriminator, vgg):

    input_low_resolution = Input(shape=(64, 64, 3))

    fake_hr_images = generator(input_low_resolution)
    fake_features = vgg(fake_hr_images)

    discriminator.trainable = False

    output = discriminator(fake_hr_images)

    model = Model(inputs=[input_low_resolution],
                  outputs=[output, fake_features])

    for layer in model.layers:
        print(layer.name, layer.trainable)
```

```
    print(model.summary())
    return model
```

这样就在 Keras 中实现了所需的网络。下面在之前下载的数据集上训练该网络。

5.5 训练 SRGAN

SRGAN 的训练分为两步。第一步，训练判别网络；第二步，训练对抗网络，即训练生成网络。下面开始训练。

训练 SRGAN 的步骤如下。

(1) 首先定义训练所需的超参数。

```
# 定义超参数
data_dir = "Paht/to/the/dataset/img_align_celeba/*.*"
epochs = 20000
batch_size = 1

# 低分辨率图像和高分辨率图像的形状
low_resolution_shape = (64, 64, 3)
high_resolution_shape = (256, 256, 3)
```

(2) 然后定义训练用的优化器。所有网络都使用 Adam 优化器，设置学习速率为 0.0002、`beta_1` 为 0.5。

```
# 所有网络共用的优化器
common_optimizer = Adam(0.0002, 0.5)
```


5.5.1 构建并编译网络

构建及编译网络的步骤如下。

(1) 构建并编译 VGG19 网络。

```
vgg = build_vgg()
vgg.trainable = False
vgg.compile(loss='mse', optimizer=common_optimizer, metrics=['accuracy'])
```

设置损失为 `mse`、度量为 `accuracy`、优化器为 `common_optimizer`，编译 VGG19。编译网络之前，将其设置为“不可训练”，因为这里不训练 VGG19 网络。

(2) 然后构建并编译判别网络，如下所示。

```
discriminator = build_discriminator()
discriminator.compile(loss='mse', optimizer=common_optimizer,
                      metrics=['accuracy'])
```

设置损失为 `mse`、度量为 `accuracy`、优化器为 `common_optimizer`，编译判别网络。

(3) 接着构建生成网络。

```
generator = build_generator()
```

(4) 然后创建对抗模型。首先创建两个输入层。

```
input_high_resolution = Input(shape=high_resolution_shape)
input_low_resolution = Input(shape=low_resolution_shape)
```

(5) 接着使用生成网络从低分辨率图像中符号式地生成高分辨率图像。

```
generated_high_resolution_images = generator(input_low_resolution)
```

使用 VGG19 从生成的图像中提取特征。

```
features = vgg(generated_high_resolution_images)
```

将判别网络设置为“不可训练”，因为训练对抗模型时不训练判别模型。

```
discriminator.trainable = False
```

(6) 然后使用判别网络获取所生成的高分辨率图像的概率。

```
probs = discriminator(generated_high_resolution_images)
```

其中，probs 表示所生成图像属于真实数据集的概率。

(7) 最后，创建并编译对抗网络。

```
adversarial_model = Model([input_low_resolution,
                          input_high_resolution], [probs, features])
adversarial_model.compile(loss=['binary_crossentropy', 'mse'],
                          loss_weights=[1e-3, 1], optimizer=common_optimizer)
```

使用 binary_crossentropy 和 mse 作为损失、common_optimizer 作为优化器、[0.001, 1]作为损失权重，编译对抗模型。

(8) 添加 TensorBoard，将训练损失和网络图可视化。

```
tensorboard = TensorBoard(log_dir="logs/".format(time.time()))
tensorboard.set_model(generator)
tensorboard.set_model(discriminator)
```

(9) 创建一个循环，运行指定轮数。

```
for epoch in range(epochs):
    print("Epoch:{}".format(epoch))
```

后续所有代码都在该循环内部。

(10) 接着采样一批高分辨率图像和低分辨率的图像，如下所示。

```
high_resolution_images, low_resolution_images = sample_images(
    data_dir=data_dir, batch_size=batch_size,low_resolution_shape=
    low_resolution_shape, high_resolution_shape=high_resolution_shape)
```

sample_images 函数的代码如下所示。这段代码详细描述了其行为。该函数按照步骤加载和缩放图像，以生成高分辨率图像和低分辨率图像。

```
def sample_images(data_dir, batch_size, high_resolution_shape,
                  low_resolution_shape):
    # 创建一个包含数据目录中所有图像的列表
    all_images = glob.glob(data_dir)

    # 随机选择一批次的图像
    images_batch = np.random.choice(all_images, size=batch_size)

    low_resolution_images = []
    high_resolution_images = []

    for img in images_batch:
        # 获取当前图像的 ndarray
        img1 = imread(img, mode='RGB')
        img1 = img1.astype(np.float32)

        # 缩放图像
        img1_high_resolution = imresize(img1, high_resolution_shape)
        img1_low_resolution = imresize(img1, low_resolution_shape)

        # 随机翻转
        if np.random.random() < 0.5:
            img1_high_resolution = np.fliplr(img1_high_resolution)
            img1_low_resolution = np.fliplr(img1_low_resolution)

        high_resolution_images.append(img1_high_resolution)
        low_resolution_images.append(img1_low_resolution)

    return np.array(high_resolution_images),
           np.array(low_resolution_images)
```

(11) 然后将图像归一化，将像素值转换为[−1, 1]区间的一个值，如下所示。

```
high_resolution_images = high_resolution_images / 127.5 - 1.
low_resolution_images = low_resolution_images / 127.5 - 1.
```

将像素值转换到−1 到 1 之间非常重要。生成网络的末端是 tanh，tanh 激活函数将数值压缩到同样的区间。对于计算损失来说，确保所有的数值都处于同样的区间很有必要。

5.5.2 训练判别网络

下面介绍训练判别网络的各个步骤。接着前面的步骤进行。

(1) 使用生成网络生成假的高分辨率图像。

```
generated_high_resolution_images = generator.predict(low_resolution_images)
```

(2) 创建一批次真标签和假标签。

5

```
real_labels = np.ones((batch_size, 16, 16, 1))
fake_labels = np.zeros((batch_size, 16, 16, 1))
```

(3) 在真图像和真标签上训练判别网络。

```
d_loss_real = discriminator.train_on_batch(high_resolution_images, real_labels)
```

(4) 在生成的图像和假标签上训练判别网络。

```
d_loss_fake = discriminator.train_on_batch(generated_high_resolution_images,
                                           fake_labels)
```

(5) 最后计算判别损失总和。

```
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

这样就添加好了训练判别网络的代码。下面添加训练对抗模型的代码，即训练生成网络的代码。

5.5.3 训练生成网络

下面介绍训练生成网络的各个步骤。接着前面的步骤进行。

(1) 再次采样一批次的高分辨率图像和低分辨率图像，并进行归一化。

```
high_resolution_images, low_resolution_images = sample_images(
    data_dir=data_dir, batch_size=batch_size,low_resolution_shape=
    low_resolution_shape, high_resolution_shape=high_resolution_shape)
# 将图像归一化
high_resolution_images = high_resolution_images / 127.5 - 1.
low_resolution_images = low_resolution_images / 127.5 - 1.
```

(2) 使用 VGG19 网络提取高分辨率真实图像的特征映射（内部表示）。

```
image_features = vgg.predict(high_resolution_images)
```

(3) 然后训练对抗模型，为其提供适当的输入，如下所示。

```
g_loss = adversarial_model.train_on_batch([low_resolution_images,
                                           high_resolution_images],
                                          [real_labels, image_features])
```

(4) 每轮训练过后，将损失写入 TensorBoard 进行可视化。

```
write_log(tensorboard, 'g_loss', g_loss[0], epoch)
write_log(tensorboard, 'd_loss', d_loss[0], epoch)
```

(5) 每训练 100 轮，使用生成网络生成假的高分辨率图像并保存，以进行可视化。

```
if epoch % 100 == 0:
    high_resolution_images, low_resolution_images = sample_images(
        data_dir=data_dir, batch_size=batch_size,low_resolution_shape=
```

```
        low_resolution_shape, high_resolution_shape=high_resolution_shape)
    # 将图像归一化
    high_resolution_images = high_resolution_images / 127.5 -1.

    low_resolution_images = low_resolution_images / 127.5 - 1.
    # 生成高分辨率假图像
    generated_images = generator.predict_on_batch(low_resolution_images)
    # 保存图像
    for index, img in enumerate(generated_images):
        save_images(low_resolution_images[index],
                    high_resolution_images[index], img, path="results/img_{}_{}".
                    format(epoch, index))
```

可以根据这些图像判定是否继续训练。如果所生成的高分辨率图像质量很好，就停止训练；否则就继续训练，直到模型足够好。

至此，SRGAN 在 CelebA 数据集上的训练就完成了，可以生成高分辨率图像了。将维度为 64×64×3 的低分辨率图像传递给 `generator.predict()` 函数，就可以生成高分辨率图像了。

5.5.4 保存模型

在 Keras 中，保存模型只需一行代码，如下所示。

```
# 指定生成网络模型的路径
gen_model.save("directory/for/the/generator/model.h5")
```

类似地，保存判别网络模型的代码如下。

```
# 指定判别网络模型的路径
dis_model.save("directory/for/the/discriminator/model.h5")
```

5.5.5 生成图像可视化

在多轮训练之后，生成网络会开始生成比较不错的图像。下面看一下这些生成的图像。

- 1000 轮训练后，图像如图 5-2 所示。

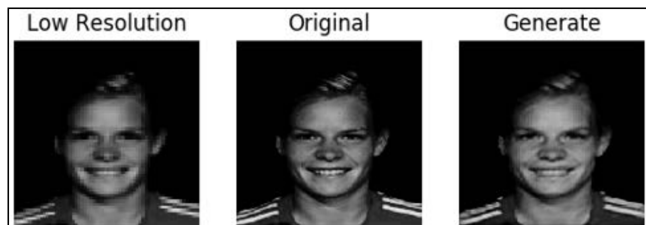


图 5-2 1000 轮训练后生成的图像

- 5000 轮训练后，图像如图 5-3 所示。



图 5-3　5000 轮训练后生成的图像

- 10 000 轮训练后，图像如图 5-4 所示。

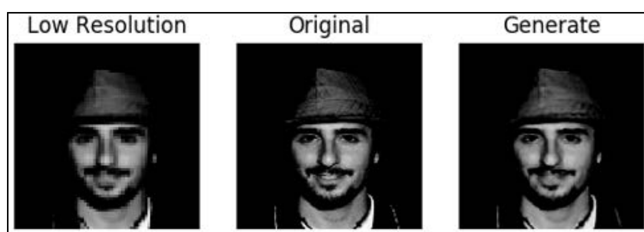


图 5-4　10 000 轮训练后生成的图像

- 15 000 轮训练后，图像如图 5-5 所示。



图 5-5　15 000 轮训练后生成的图像

- 20 000 轮训练后，图像如图 5-6 所示。

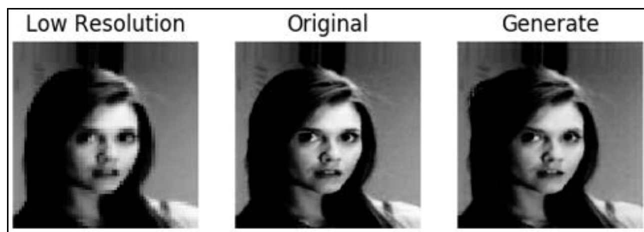


图 5-6　20 000 轮训练后生成的图像

将网络训练 30 000~50 000 轮，可生成非常不错的图像。

5.5.6 损失可视化

启动 TensorBoard 服务器，将训练损失可视化。如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图，如图 5-7 所示。

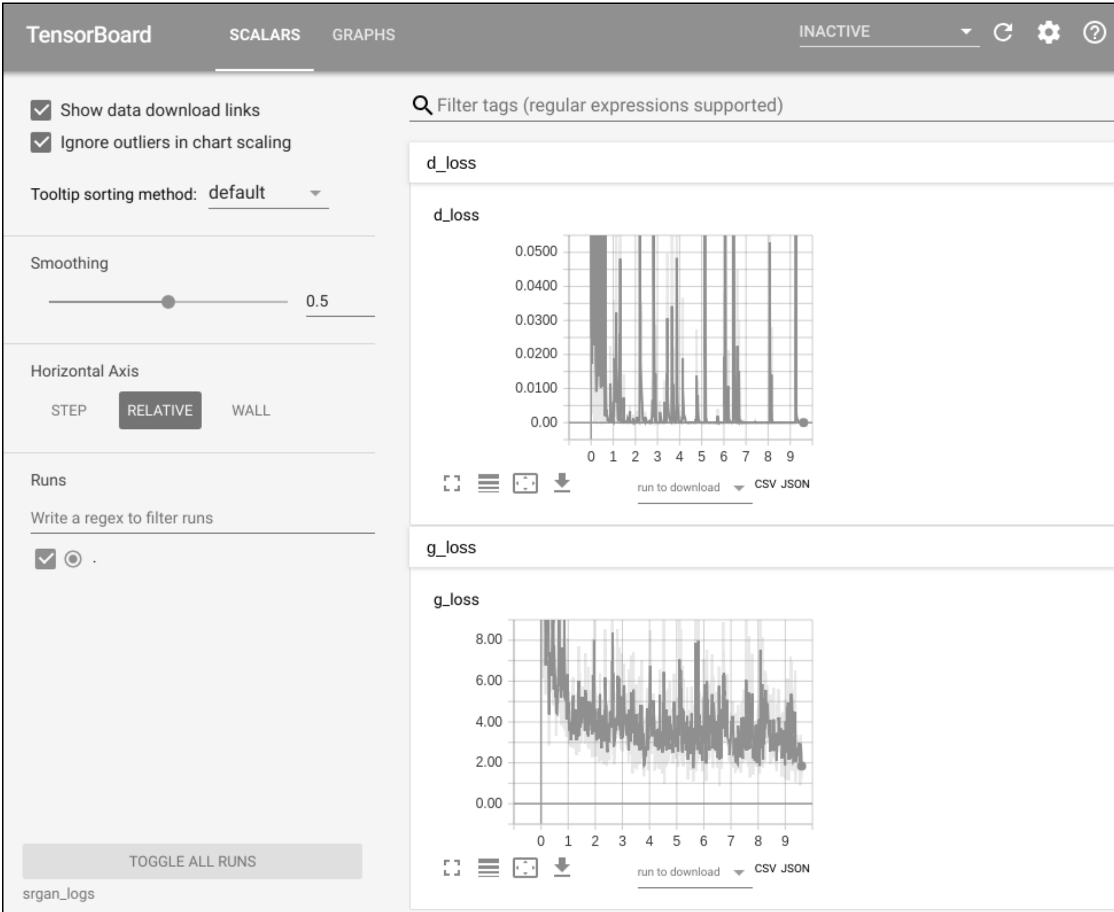


图 5-7 两种损失的曲线图

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

5.5.7 图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 5-8）。

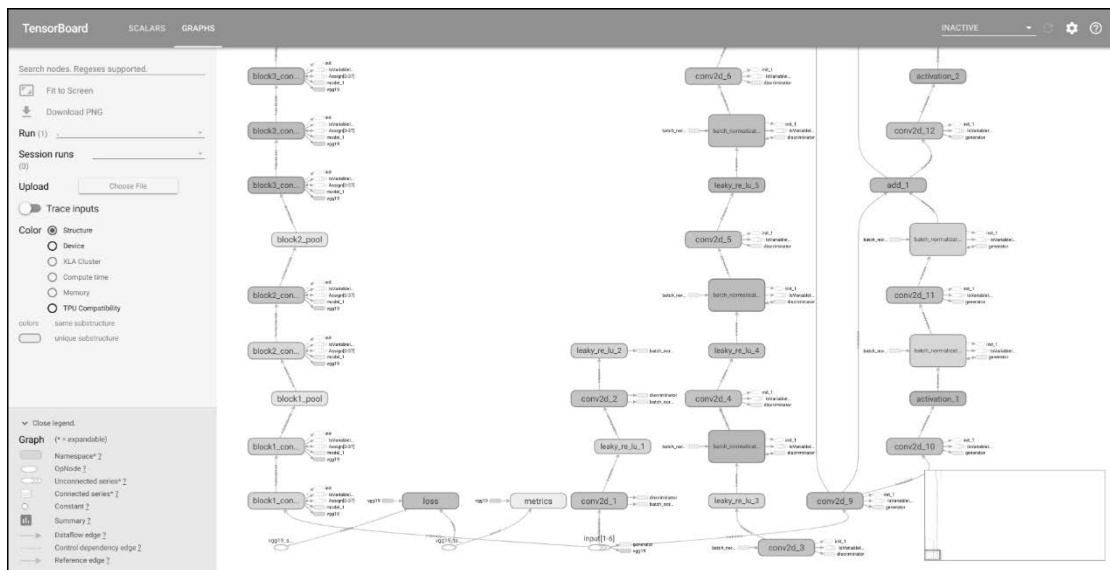


图 5-8 各图中的张量和不同运算的流

5.6 SRGAN 的实际应用

SRGAN 的实际应用包括：

- 修复旧照片；
- 行业应用，比如自动提高 logo、条幅和宣传页的分辨率；
- 为用户自动提高社交媒体图像的分辨率；
- 使用相机拍摄照片时自动增强图像；
- 提高医学图像的分辨率。

5.7 小结

本章首先介绍了 SRGAN 以及其生成网络和判别网络的架构，然后创建了项目，收集并探索了数据集，接着使用 Keras 实现了项目，训练了 SRGAN 并评估了其表现，最后简单介绍了 SRGAN 的各种应用。

下一章将介绍 StackGAN 及其应用。

第 6 章

StackGAN：基于文本合成逼真图像

6

基于文本合成图像是 GAN 的用途之一，和前面几章介绍的 GAN 类似，它具有广泛的行业应用。基于文本合成图像非常复杂，因为能生成反映文本意义的图像模型不易构建。StackGAN 就是为解决该问题而设计的。本章使用以 TensorFlow 为后端的 Keras 框架，实现一个 StackGAN。

本章将讨论以下主题。

- StackGAN 简介
- StackGAN 架构
- 数据收集和准备
- StackGAN 的 Keras 实现
- 训练 StackGAN
- 模型评估
- StackGAN 的实际应用

6.1 StackGAN 简介

如名所示，StackGAN 是由两个 GAN 堆叠而成的可生成高分辨率图像的网络。StackGAN 包含两个阶段：第一阶段和第二阶段。第一阶段网络以文本嵌入为条件，生成具有基础颜色和框架的低分辨率图像；第二阶段网络接收第一阶段网络生成的图像，然后以文本嵌入为条件，生成高分辨率图像。简单说来，第二个网络会修正一些缺陷，并添加一些令人信服的细节，以获得更为逼真的高分辨率图像。

StackGAN 的工作方式类似于画家。画家作画时，首先会绘制一些基础的形状，比如线条、圆形和矩形；接着，画家会填充颜色；随着绘画过程的推进，添加更多细节。在 StackGAN 中，第一阶段绘制基础形状，第二阶段修正第一阶段网络生成的图像中的缺陷，并 V 添加一些细节，使图像看起来更逼真。两个阶段的生成网络都是 cGAN。第一个 GAN 以文本描述为条件，第二

个以文本描述和第一个 GAN 生成的图像为条件。

6.2 StackGAN 架构

StackGAN 是一个二阶段网络，每个阶段都有一个生成网络和一个判别网络。StackGAN 的网络构成如下所示。

- 第一阶段 GAN：文本编码网络、条件增强网络、生成网络、判别网络和嵌入压缩网络。
- 第二阶段 GAN：文本编码网络、条件增强网络、生成网络、判别网络和嵌入压缩网络。

图 6-1 展示了 StackGAN 的两个阶段，比较直观易懂。其中第一阶段生成维度为 64×64 的图像，然后第二阶段接收这些低分辨率图像，然后生成维度为 256×256 的高分辨率图像。下面详细介绍 StackGAN 的各组件。在此之前，首先介绍本章使用的各种记法（见表 6-1）。

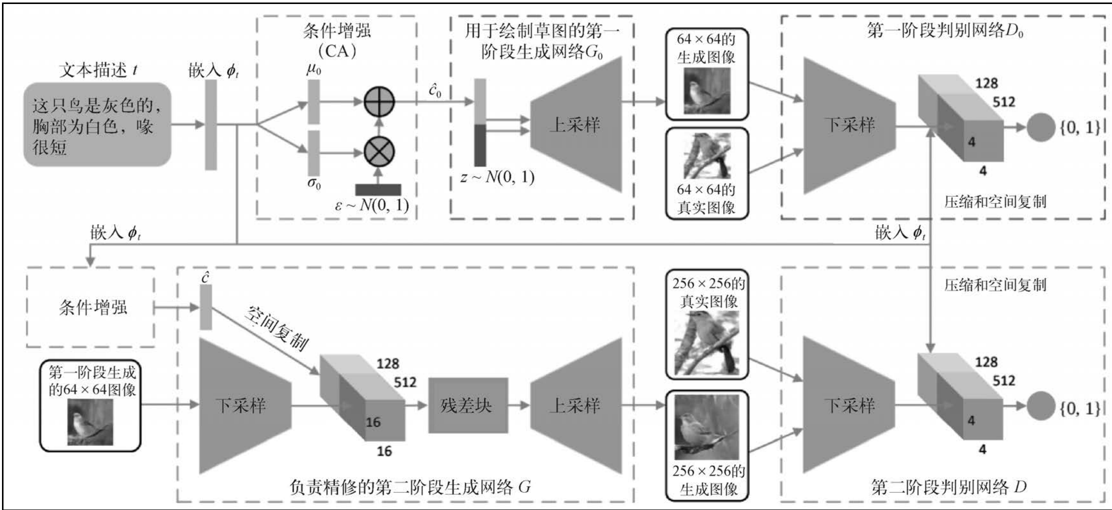


图 6-1 StackGAN 的两个阶段

表 6-1

记　法	描　述
t	对真实数据分布的一段文本描述
z	从高斯分布中随机采样的一个噪声向量
ϕ_t	预训练编码网络基于给定文本生成的一个文本嵌入
$\hat{c}_0$	该文本条件变量是从概率分布 $N(\mu_0(\phi_t),\Sigma_0(\phi_t))$ 中采样的一个高斯条件变量，它捕捉到了 ϕ_t 的不同含义
$N(\mu_0(\phi_t),\Sigma(\phi_t))$	一个条件高斯分布
$N(0, 1)$	一个正态分布

（续）

记　法	描　述
$\Sigma(\phi_t)$	一个对角协方差矩阵
p_{data}	真实数据分布
p_z	高斯分布
d_1	第一阶段判别网络
g_1	第一阶段生成网络
d_2	第二阶段判别网络
g_2	第二阶段生成网络
n_2	随机噪声变量的维度
$\hat{c}$	第二阶段 GAN 的高斯潜在变量

6.2.1 文本编码网络

文本编码网络仅负责将文本描述（t）转换为文本嵌入（ϕ_t）。本章不会训练该文本编码网络，而会使用预训练的文本嵌入。如前所示准备数据，下载预训练的文本嵌入。如果想训练自己的文本编码网络，请参考论文“Learning Deep Representations of Fine-Grained Visual Descriptions”。文本编码网络将一句话编码为一个 1024 维的文本嵌入。两个阶段使用相同的文本编码网络。

6.2.2 CA 块

CA 网络从概率分布 $N(\mu(\phi_t),\Sigma(\phi_t))$ 中采样随机潜在变量 $\hat{c}$。后面详细介绍该分布。添加 CA 块有很多好处，如下所示。

- 为网络增添随机性。
- 通过捕捉不同对象的不同姿势和形象，让生成网络变得稳健。
- 产生更多的“图像-文本”对。如果“图像-文本”对的数量足够多，就可以训练出能应对扰动的稳健网络。

获取 CA 变量

通过文本编码网络得到文本嵌入（ϕ_t后，需要将其传递给一个全连接层，以生成一些数值，包括平均值 μ_0 和标准差 σ_0。然后使用这些数值创建对角协方差矩阵，并将 σ_0 放在矩阵（$\Sigma(\phi_t)$）的对角线上。最后，使用 μ_0 和 Σ_0 创建高斯分布，形式如下。

$$N(\mu_0(\phi_t),\Sigma_0(\phi_t))$$

然后从刚刚创建的高斯分布中采样 $\hat{c}_0$。$\hat{c}_0$ 的计算公式如下。

$$\hat{c}_0 = \mu_0 + \sigma_0 \odot N(0, I)$$

上式浅显易懂。采样 $\hat{c}_0$ 的方式是，首先与 σ_0 的逐元素相乘，再将结果与 μ_0 相加。6.5 节中会详细介绍 CA 变量 $\hat{c}_0$ 的计算方法。

6.2.3　第一阶段

StackGAN 主要由生成网络和判别网络组成。下面详细介绍这两个网络。

1. 生成网络

第一阶段的生成网络是一个具有多个上采样层的深度卷积神经网络。该生成网络是一个 cGAN，以 $\hat{c}_0$ 和随机变量 z 为条件。生成网络接收高斯条件变量 $\hat{c}_0$ 和随机噪声变量 z，然后生成维度为 64×64×3 的图像。其生成的低分辨率图像可能已经具有了基础形状和颜色，但仍有各种缺陷。其中 z 是从高斯分布 p_z 中采样的维度为 N_z 的随机噪声变量。生成网络生成的图像可表示为 $s_0 = G_0(z, \hat{c}_0)$。生成网络的架构如图 6-2 所示。

```
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
input_1 (InputLayer)            (None, 1024)         0
____________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          262400      input_1[0][0]
____________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 256)          0           dense_1[0][0]
____________________________________________________________________________________________________
lambda_1 (Lambda)               (None, 128)          0           leaky_re_lu_1[0][0]
____________________________________________________________________________________________________
input_2 (InputLayer)            (None, 100)          0
____________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 228)          0           lambda_1[0][0]
                                                                 input_2[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                 (None, 16384)        3735552     concatenate_1[0][0]
____________________________________________________________________________________________________
re_lu_1 (ReLU)                  (None, 16384)        0           dense_2[0][0]
____________________________________________________________________________________________________
reshape_1 (Reshape)             (None, 4, 4, 1024)   0           re_lu_1[0][0]
____________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 8, 8, 1024)   0           reshape_1[0][0]
____________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 8, 8, 512)    4718592     up_sampling2d_1[0][0]
____________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 8, 8, 512)    2048        conv2d_1[0][0]
____________________________________________________________________________________________________
re_lu_2 (ReLU)                  (None, 8, 8, 512)    0           batch_normalization_1[0][0]
____________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 16, 16, 512)  0           re_lu_2[0][0]
____________________________________________________________________________________________________
```

图 6-2　第一阶段的生成网络架构

```
____________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 16, 16, 256)  1179648     up_sampling2d_2[0][0]
____________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 16, 16, 256)  1024        conv2d_2[0][0]
____________________________________________________________________________________________________
re_lu_3 (ReLU)                  (None, 16, 16, 256)  0           batch_normalization_2[0][0]
____________________________________________________________________________________________________
up_sampling2d_3 (UpSampling2D)  (None, 32, 32, 256)  0           re_lu_3[0][0]
____________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 32, 32, 128)  294912      up_sampling2d_3[0][0]
____________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 32, 32, 128)  512         conv2d_3[0][0]
____________________________________________________________________________________________________
re_lu_4 (ReLU)                  (None, 32, 32, 128)  0           batch_normalization_3[0][0]
____________________________________________________________________________________________________
up_sampling2d_4 (UpSampling2D)  (None, 64, 64, 128)  0           re_lu_4[0][0]
____________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 64, 64, 64)   73728       up_sampling2d_4[0][0]
____________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 64, 64, 64)   256         conv2d_4[0][0]
____________________________________________________________________________________________________
re_lu_5 (ReLU)                  (None, 64, 64, 64)   0           batch_normalization_4[0][0]
____________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 64, 64, 3)    1728        re_lu_5[0][0]
____________________________________________________________________________________________________
activation_1 (Activation)       (None, 64, 64, 3)    0           conv2d_5[0][0]
====================================================================================================
Total params: 10,270,400
Trainable params: 10,268,480
Non-trainable params: 1,920
____________________________________________________________________________________________________
```

图 6-2 （续）

如上所示，生成网络包含多个卷积层，每个卷积层后面都跟着一个批归一化层或激活层。生成网络仅负责生成维度为 64×64×3 的图像。介绍过了生成网络，下面介绍判别网络。

6

2. 判别网络

和生成网络类似，判别网络是深度卷积神经网络，包含一系列下采样卷积层。下采样层生成图像的特征映射，包括遵循真实数据分布的真实图像和生成网络生成的图像。然后将特征映射和文本嵌入拼接在一起。通过压缩和空间复制将文本嵌入转换成拼接所需的形式。压缩和空间复制过程首先包括一个全连接层，用于将文本嵌入压缩为 N_d 维度的输出，再通过空间复制转换为维度为 $M_d \times M_d \times N_d$ 的张量，接着沿通道维度将特征映射和经过压缩和空间复制的文本嵌入拼接在一起。最后，使用一个含一个节点的全连接层进行二分类。判别网络的架构如图 6-3 所示。

```
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
input_3 (InputLayer)            (None, 64, 64, 3)    0
____________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 32, 32, 64)   3072        input_3[0][0]
____________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 32, 32, 64)   0           conv2d_6[0][0]
____________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 16, 16, 128)  131072      leaky_re_lu_2[0][0]
____________________________________________________________________________________________________
batch_normalization_5 (BatchNor (None, 16, 16, 128)  512         conv2d_7[0][0]
____________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)       (None, 16, 16, 128)  0           batch_normalization_5[0][0]
____________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 8, 8, 256)    524288      leaky_re_lu_3[0][0]
____________________________________________________________________________________________________
batch_normalization_6 (BatchNor (None, 8, 8, 256)    1024        conv2d_8[0][0]
____________________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)       (None, 8, 8, 256)    0           batch_normalization_6[0][0]
____________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 4, 4, 512)    2097152     leaky_re_lu_4[0][0]
____________________________________________________________________________________________________
batch_normalization_7 (BatchNor (None, 4, 4, 512)    2048        conv2d_9[0][0]
____________________________________________________________________________________________________
leaky_re_lu_5 (LeakyReLU)       (None, 4, 4, 512)    0           batch_normalization_7[0][0]
____________________________________________________________________________________________________
input_4 (InputLayer)            (None, 4, 4, 128)    0
____________________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 4, 4, 640)    0           leaky_re_lu_5[0][0]
                                                                 input_4[0][0]
____________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 4, 4, 512)    328192      concatenate_2[0][0]
____________________________________________________________________________________________________
batch_normalization_8 (BatchNor (None, 4, 4, 512)    2048        conv2d_10[0][0]
____________________________________________________________________________________________________
leaky_re_lu_6 (LeakyReLU)       (None, 4, 4, 512)    0           batch_normalization_8[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 8192)         0           leaky_re_lu_6[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                 (None, 1)            8193        flatten_1[0][0]
____________________________________________________________________________________________________
activation_2 (Activation)       (None, 1)            0           dense_3[0][0]
====================================================================================================
Total params: 3,097,601
Trainable params: 3,094,785
Non-trainable params: 2,816
____________________________________________________________________________________________________
```

图 6-3　第一阶段的判别网络架构

3. StackGAN 第一阶段的损失

StackGAN 第一阶段使用了两种损失，分别是：

- 生成网络损失

❑ 判别网络损失

判别网络损失 L_D形式如下。

$$L_{D_0} = \mathbb{E}_{(I_0,t)\sim p_{\text{data}}}[\log D_0(I_0,\phi_t)] + \mathbb{E}_{z\sim p_z,t\sim p_{\text{data}}}[\log(1-D_0(G_0(z,\hat{c}_0),\phi_t))]$$

上式清楚地表示了判别网络的损失函数，其中两个网络都以文本嵌入为条件。

生成网络损失 L_G形式如下。

$$L_{G_0} = \mathbb{E}_{z\sim p_z,t\sim p_{\text{data}}}[\log(1-D_0(G_0(z,\hat{c}_0),\phi_t))] + \lambda D_{\text{KL}}(N(\mu_0(\phi_t),\Sigma_0(\phi_t)) \,\|\, N(0,I))$$

上式清楚地表示了生成网络的损失函数，其中两个网络都以文本嵌入为条件。损失函数中还包含了一个 KL 散度项。

6.2.4 第二阶段

StackGAN 第二阶段主要由一个生成网络和一个判别网络组成。生成网络是一种编码解码网络。该阶段不会使用随机噪声 z，因为在 s_0（第一阶段生成网络生成的图像）中保留了假设随机性。

首先使用预训练的文本编码网络生成高斯条件变量 $\hat{c}_0$，这会生成相同的文本嵌入 ϕ_t。第一阶段和第二阶段的条件增强网络的全连接层不同，以生成不同的平均值和标准差。这样第二阶段 GAN 就可以学习捕捉文本嵌入中被第一阶段 GAN 忽略的有用信息了。

第一阶段 GAN 生成图像的问题是不够生动。这些图像中可能包含扭曲了的形状，也可能忽略了一些重要的细节，而这些细节对于生成逼真图像来说十分关键。第二阶段 GAN 基于第一阶段 GAN 的输出进行构建，以第一阶段 GAN 生成的低分辨率图像和文本描述为条件，修正缺陷以生成高分辨率图像。

1. 生成网络

生成网络也是深度卷积神经网络。第一阶段输出的结果（即低分辨率图像）经过几个下采样层传递，输出图像特征。接沿通道维度拼接图像特征和文本条件变量，然后将拼接好的张量传递给一些残差块，以学习图像和文本特征的多模式表示。最后，将上一步操作的输出传递给一组上采样层，输出维度为 256×256×3 的高分辨率图像。生成网络的架构如图 6-4 所示。

```
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
input_2 (InputLayer)            (None, 64, 64, 3)    0
____________________________________________________________________________________________________
zero_padding2d_1 (ZeroPadding2D (None, 66, 66, 3)    0           input_2[0][0]
____________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 64, 64, 128)  3456        zero_padding2d_1[0][0]
____________________________________________________________________________________________________
re_lu_1 (ReLU)                  (None, 64, 64, 128)  0           conv2d_1[0][0]
____________________________________________________________________________________________________
zero_padding2d_2 (ZeroPadding2D (None, 66, 66, 128)  0           re_lu_1[0][0]
____________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 32, 32, 256)  524288      zero_padding2d_2[0][0]
____________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 32, 32, 256)  1024        conv2d_2[0][0]
____________________________________________________________________________________________________
re_lu_2 (ReLU)                  (None, 32, 32, 256)  0           batch_normalization_1[0][0]
____________________________________________________________________________________________________
input_1 (InputLayer)            (None, 1024)         0
____________________________________________________________________________________________________
zero_padding2d_3 (ZeroPadding2D (None, 34, 34, 256)  0           re_lu_2[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          262400      input_1[0][0]
____________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 16, 16, 512)  2097152     zero_padding2d_3[0][0]
____________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 256)          0           dense_1[0][0]
____________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 16, 16, 512)  2048        conv2d_3[0][0]
____________________________________________________________________________________________________
lambda_1 (Lambda)               (None, 128)          0           leaky_re_lu_1[0][0]
____________________________________________________________________________________________________
re_lu_3 (ReLU)                  (None, 16, 16, 512)  0           batch_normalization_2[0][0]
____________________________________________________________________________________________________
lambda_2 (Lambda)               (None, 16, 16, 640)  0           lambda_1[0][0]
                                                                 re_lu_3[0][0]
____________________________________________________________________________________________________
zero_padding2d_4 (ZeroPadding2D (None, 18, 18, 640)  0           lambda_2[0][0]
____________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 16, 16, 512)  2949120     zero_padding2d_4[0][0]
____________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 16, 16, 512)  2048        conv2d_4[0][0]
____________________________________________________________________________________________________
re_lu_4 (ReLU)                  (None, 16, 16, 512)  0           batch_normalization_3[0][0]
____________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 16, 16, 512)  2359808     re_lu_4[0][0]
____________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 16, 16, 512)  2048        conv2d_5[0][0]
____________________________________________________________________________________________________
re_lu_5 (ReLU)                  (None, 16, 16, 512)  0           batch_normalization_4[0][0]
____________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 16, 16, 512)  2359808     re_lu_5[0][0]
____________________________________________________________________________________________________
batch_normalization_5 (BatchNor (None, 16, 16, 512)  2048        conv2d_6[0][0]
____________________________________________________________________________________________________
add_1 (Add)                     (None, 16, 16, 512)  0           batch_normalization_5[0][0]
                                                                 re_lu_4[0][0]
____________________________________________________________________________________________________
re_lu_6 (ReLU)                  (None, 16, 16, 512)  0           add_1[0][0]
____________________________________________________________________________________________________
```

图 6-4 第二阶段的生成网络架构

```
________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 16, 16, 512)  2359808     re_lu_6[0][0]
________________________________________________________________________________________________
batch_normalization_6 (BatchNor (None, 16, 16, 512)  2048        conv2d_7[0][0]
________________________________________________________________________________________________
re_lu_7 (ReLU)                  (None, 16, 16, 512)  0           batch_normalization_6[0][0]
________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 16, 16, 512)  2359808     re_lu_7[0][0]
________________________________________________________________________________________________
batch_normalization_7 (BatchNor (None, 16, 16, 512)  2048        conv2d_8[0][0]
________________________________________________________________________________________________
add_2 (Add)                     (None, 16, 16, 512)  0           batch_normalization_7[0][0]
                                                                 re_lu_6[0][0]
________________________________________________________________________________________________
re_lu_8 (ReLU)                  (None, 16, 16, 512)  0           add_2[0][0]
________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 16, 16, 512)  2359808     re_lu_8[0][0]
________________________________________________________________________________________________
batch_normalization_8 (BatchNor (None, 16, 16, 512)  2048        conv2d_9[0][0]
________________________________________________________________________________________________
re_lu_9 (ReLU)                  (None, 16, 16, 512)  0           batch_normalization_8[0][0]
________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 16, 16, 512)  2359808     re_lu_9[0][0]
________________________________________________________________________________________________
batch_normalization_9 (BatchNor (None, 16, 16, 512)  2048        conv2d_10[0][0]
________________________________________________________________________________________________
add_3 (Add)                     (None, 16, 16, 512)  0           batch_normalization_9[0][0]
                                                                 re_lu_8[0][0]
________________________________________________________________________________________________
re_lu_10 (ReLU)                 (None, 16, 16, 512)  0           add_3[0][0]
________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 16, 16, 512)  2359808     re_lu_10[0][0]
________________________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 16, 16, 512)  2048        conv2d_11[0][0]
________________________________________________________________________________________________
re_lu_11 (ReLU)                 (None, 16, 16, 512)  0           batch_normalization_10[0][0]
________________________________________________________________________________________________
conv2d_12 (Conv2D)              (None, 16, 16, 512)  2359808     re_lu_11[0][0]
________________________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 16, 16, 512)  2048        conv2d_12[0][0]
________________________________________________________________________________________________
add_4 (Add)                     (None, 16, 16, 512)  0           batch_normalization_11[0][0]
                                                                 re_lu_10[0][0]
________________________________________________________________________________________________
re_lu_12 (ReLU)                 (None, 16, 16, 512)  0           add_4[0][0]
________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 32, 32, 512)  0           re_lu_12[0][0]
________________________________________________________________________________________________
conv2d_13 (Conv2D)              (None, 32, 32, 512)  2359296     up_sampling2d_1[0][0]
________________________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 32, 32, 512)  2048        conv2d_13[0][0]
________________________________________________________________________________________________
re_lu_13 (ReLU)                 (None, 32, 32, 512)  0           batch_normalization_12[0][0]
________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 64, 64, 512)  0           re_lu_13[0][0]
________________________________________________________________________________________________
conv2d_14 (Conv2D)              (None, 64, 64, 256)  1179648     up_sampling2d_2[0][0]
________________________________________________________________________________________________
batch_normalization_13 (BatchNo (None, 64, 64, 256)  1024        conv2d_14[0][0]
________________________________________________________________________________________________
re_lu_14 (ReLU)                 (None, 64, 64, 256)  0           batch_normalization_13[0][0]
________________________________________________________________________________________________
```

图 6-4 （续）

```
____________________________________________________________________________________________________
up_sampling2d_3 (UpSampling2D)  (None, 128, 128, 256 0           re_lu_14[0][0]
____________________________________________________________________________________________________
conv2d_15 (Conv2D)              (None, 128, 128, 128 294912      up_sampling2d_3[0][0]
____________________________________________________________________________________________________
batch_normalization_14 (BatchNo (None, 128, 128, 128 512         conv2d_15[0][0]
____________________________________________________________________________________________________
re_lu_15 (ReLU)                 (None, 128, 128, 128 0           batch_normalization_14[0][0]
____________________________________________________________________________________________________
up_sampling2d_4 (UpSampling2D)  (None, 256, 256, 128 0           re_lu_15[0][0]
____________________________________________________________________________________________________
conv2d_16 (Conv2D)              (None, 256, 256, 64) 73728       up_sampling2d_4[0][0]
____________________________________________________________________________________________________
batch_normalization_15 (BatchNo (None, 256, 256, 64) 256         conv2d_16[0][0]
____________________________________________________________________________________________________
re_lu_16 (ReLU)                 (None, 256, 256, 64) 0           batch_normalization_15[0][0]
____________________________________________________________________________________________________
conv2d_17 (Conv2D)              (None, 256, 256, 3)  1728        re_lu_16[0][0]
____________________________________________________________________________________________________
activation_1 (Activation)       (None, 256, 256, 3)  0           conv2d_17[0][0]
====================================================================================================
Total params: 28,649,536
Trainable params: 28,636,864
Non-trainable params: 12,672
____________________________________________________________________________________________________
```

图 6-4　（续）

该生成网络仅负责使用低分辨率图像生成高分辨率图像。第一阶段生成网络首先生成低分辨率图像，然后传递给第二阶段生成网络来生成高分辨率图像。

2. 判别网络

和生成网络类似，判别网络也是深度卷积神经网络，但包含额外的下采样层，因为输入图像比第一阶段中判别网络的输入图像更大。该判别网络是一个匹配察觉（matching-aware）判别网络，有助于实现与条件文本更对应的图像。训练时，判别网络接收真实图像和对应的文本描述，作为一对正样本。负样本由两组构成，第一组是真实图像和不匹配的文本嵌入，第二组是合成图像以及对应的文本嵌入。判别网络的架构如图 6-5 所示。

```
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
input_3 (InputLayer)            (None, 256, 256, 3)  0
____________________________________________________________________________________________________
conv2d_18 (Conv2D)              (None, 128, 128, 64) 3072        input_3[0][0]
____________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 128, 128, 64) 0           conv2d_18[0][0]
____________________________________________________________________________________________________
conv2d_19 (Conv2D)              (None, 64, 64, 128)  131072      leaky_re_lu_2[0][0]
____________________________________________________________________________________________________
batch_normalization_16 (BatchNo (None, 64, 64, 128)  512         conv2d_19[0][0]
____________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)       (None, 64, 64, 128)  0           batch_normalization_16[0][0]
____________________________________________________________________________________________________
conv2d_20 (Conv2D)              (None, 32, 32, 256)  524288      leaky_re_lu_3[0][0]
____________________________________________________________________________________________________
```

图 6-5　第二阶段的判别网络架构

```
____________________________________________________________________________________________________
batch_normalization_17 (BatchNo (None, 32, 32, 256)  1024        conv2d_20[0][0]
____________________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)       (None, 32, 32, 256)  0           batch_normalization_17[0][0]
____________________________________________________________________________________________________
conv2d_21 (Conv2D)              (None, 16, 16, 512)  2097152     leaky_re_lu_4[0][0]
____________________________________________________________________________________________________
batch_normalization_18 (BatchNo (None, 16, 16, 512)  2048        conv2d_21[0][0]
____________________________________________________________________________________________________
leaky_re_lu_5 (LeakyReLU)       (None, 16, 16, 512)  0           batch_normalization_18[0][0]
____________________________________________________________________________________________________
conv2d_22 (Conv2D)              (None, 8, 8, 1024)   8388608     leaky_re_lu_5[0][0]
____________________________________________________________________________________________________
batch_normalization_19 (BatchNo (None, 8, 8, 1024)   4096        conv2d_22[0][0]
____________________________________________________________________________________________________
leaky_re_lu_6 (LeakyReLU)       (None, 8, 8, 1024)   0           batch_normalization_19[0][0]
____________________________________________________________________________________________________
conv2d_23 (Conv2D)              (None, 4, 4, 2048)   33554432    leaky_re_lu_6[0][0]
____________________________________________________________________________________________________
batch_normalization_20 (BatchNo (None, 4, 4, 2048)   8192        conv2d_23[0][0]
____________________________________________________________________________________________________
leaky_re_lu_7 (LeakyReLU)       (None, 4, 4, 2048)   0           batch_normalization_20[0][0]
____________________________________________________________________________________________________
conv2d_24 (Conv2D)              (None, 4, 4, 1024)   2097152     leaky_re_lu_7[0][0]
____________________________________________________________________________________________________
batch_normalization_21 (BatchNo (None, 4, 4, 1024)   4096        conv2d_24[0][0]
____________________________________________________________________________________________________
leaky_re_lu_8 (LeakyReLU)       (None, 4, 4, 1024)   0           batch_normalization_21[0][0]
____________________________________________________________________________________________________
conv2d_25 (Conv2D)              (None, 4, 4, 512)    524288      leaky_re_lu_8[0][0]
____________________________________________________________________________________________________
batch_normalization_22 (BatchNo (None, 4, 4, 512)    2048        conv2d_25[0][0]
____________________________________________________________________________________________________
conv2d_26 (Conv2D)              (None, 4, 4, 128)    65536       batch_normalization_22[0][0]
____________________________________________________________________________________________________
batch_normalization_23 (BatchNo (None, 4, 4, 128)    512         conv2d_26[0][0]
____________________________________________________________________________________________________
leaky_re_lu_9 (LeakyReLU)       (None, 4, 4, 128)    0           batch_normalization_23[0][0]
____________________________________________________________________________________________________
conv2d_27 (Conv2D)              (None, 4, 4, 128)    147456      leaky_re_lu_9[0][0]
____________________________________________________________________________________________________
batch_normalization_24 (BatchNo (None, 4, 4, 128)    512         conv2d_27[0][0]
____________________________________________________________________________________________________
leaky_re_lu_10 (LeakyReLU)      (None, 4, 4, 128)    0           batch_normalization_24[0][0]
____________________________________________________________________________________________________
conv2d_28 (Conv2D)              (None, 4, 4, 512)    589824      leaky_re_lu_10[0][0]
____________________________________________________________________________________________________
batch_normalization_25 (BatchNo (None, 4, 4, 512)    2048        conv2d_28[0][0]
____________________________________________________________________________________________________
add_5 (Add)                     (None, 4, 4, 512)    0           batch_normalization_22[0][0]
                                                                 batch_normalization_25[0][0]
____________________________________________________________________________________________________
leaky_re_lu_11 (LeakyReLU)      (None, 4, 4, 512)    0           add_5[0][0]
____________________________________________________________________________________________________
input_4 (InputLayer)            (None, 4, 4, 128)    0
____________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 4, 4, 640)    0           leaky_re_lu_11[0][0]
                                                                 input_4[0][0]
____________________________________________________________________________________________________
conv2d_29 (Conv2D)              (None, 4, 4, 512)    328192      concatenate_1[0][0]
____________________________________________________________________________________________________
```

图 6-5 （续）

```
----------------------------------------------------------------------------------------------------
batch_normalization_26 (BatchNo (None, 4, 4, 512)    2048        conv2d_29[0][0]
----------------------------------------------------------------------------------------------------
leaky_re_lu_12 (LeakyReLU)      (None, 4, 4, 512)    0           batch_normalization_26[0][0]
----------------------------------------------------------------------------------------------------
flatten_1 (Flatten)             (None, 8192)         0           leaky_re_lu_12[0][0]
----------------------------------------------------------------------------------------------------
dense_2 (Dense)                 (None, 1)            8193        flatten_1[0][0]
----------------------------------------------------------------------------------------------------
activation_2 (Activation)       (None, 1)            0           dense_2[0][0]
====================================================================================================
Total params: 48,486,401
Trainable params: 48,472,833
Non-trainable params: 13,568

----------------------------------------------------------------------------------------------------
```

图 6-5　（续）

6.5 节会详细介绍判别网络的架构。

3. 第二阶段 StackGAN 的损失

和其他 GAN 类似，第二阶段 GAN 的生成网络 G 和判别网络 D 也可以通过将判别网络的损失最大化、生成网络的损失最小化而进行训练。

判别网络损失 L_D 形式如下。

$$L_D = \mathbb{E}_{(I,t)\sim p_{\text{data}}}[\log D(I,\phi_t)] + \mathbb{E}_{s_0\sim pG_0, t\sim p_{\text{data}}}[\log(1-D(G(s_0,\hat{c}),\phi_t))]$$

上式清楚地表示了判别网络的损失函数，其中两个网络都以文本嵌入为条件。一个主要的区别是，生成网络的输入是 s_0 和 $\hat{c}$，其中 s_0 是第一阶段生成的图像，$\hat{c}$ 是 CA 变量。

生成网络损失 L_G 形式如下。

$$L_G = \mathbb{E}_{s_0\sim pG_0, t\sim p_{\text{data}}}[\log(1-D(G(s_0,\hat{c}),\phi_t))] + \lambda D_{\text{KL}}(N(\mu(\phi_t),\Sigma(\phi_t)) \,\|\, N(0,I))$$

上式清楚地表示了生成网络的损失函数，其中两个网络都以文本嵌入为条件。损失函数中还包含了一个 KL 散度项。

6.3　创建项目

前面已经克隆或下载了本书所有章节的完整代码。其中目录 Chapter06 包含本章的完整代码。执行如下命令以创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter06。

```
cd Chapter06
```

(3) 然后为本项目创建一个 Python 虚拟环境。

```
virtualenv venv
virtualenv venv -p python3 # 创建一个使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建一个使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 接着启用新创建的虚拟环境。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 然后执行如下命令，安装 requirements.txt 文件中列出的所需的库。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

这样就创建好了项目并安装了所需的依赖程序。下面处理数据集。

6.4　准备数据

本节使用 CUB 数据集，该数据集包含不同鸟类的高分辨率图片，共 11 788 张。此外，需要 char-CNN-RNN 文本嵌入，这是预训练的文本嵌入。请按照如下指引下载并提取数据集。

6

6.4.1　下载数据集

可以手动下载 CUB 数据集，地址为：http://www.vision.caltech.edu/visipedia/CUB-200-2011.html。或者执行以下命令下载数据集。

```
Wget
http://www.vision.caltech.edu/visipedia-data/CUB-200-2011/CUB_200_2011.tgz
```

然后提取数据集至 data/birds/目录下。

访问 https://drive.google.com/open?id=0B3y_msrWZaxLT1BZdVdycDY5TEE 下载 char-CNN-RNN 嵌入。

6.4.2　提取数据集

CUB 数据集是压缩文件，提取命令如下。

```
tar -xvzf CUB_200_2011.tgz
```

使用以下命令提取 char-CNN-RNN 嵌入。

```
unzip birds.zip
```

最后，将 CUB_200_2011 放在 data/birds 目录中。至此，数据集就准备好了。

6.4.3　探索数据集

CUB 数据集包含 200 种鸟类的图片，共 11 788 张。图 6-6 展示了其中几张。



图 6-6　几张鸟类图像

这 4 张图展示的分别是一只黑脚信天翁、一只白腹小海鹦、一只食米鸟和一只勃兰特鸬鹚。

在着手设计网络之前，需要先理解数据集。请务必仔细浏览数据集中的图像。

6.5　StackGAN 的 Keras 实现

StackGAN 的 Keras 实现分为两部分：第一阶段和第二阶段，下面依次实现。

6.5.1　第一阶段

StackGAN 的第一阶段包含一个生成网络、一个判别网络、一个文本编码网络和一个 CA 网络，稍后具体讲解。生成网络接收文本条件变量（$\hat{c}_0$）以及噪声向量（$\boldsymbol{x}$），经过一系列上采样层之后，

生成维度为 64×64×3 的低分辨率图像。判别网络接收该低分辨率图像，试图分辨该图像的真假。判别网络有一系列下采样层，最后是一个拼接层和一个分类层。下面详细介绍 StackGAN 的架构。

StackGAN 的第一阶段使用的网络如下所示。

- 一个文本编码网络
- 一个 CA 网络
- 一个生成网络
- 一个判别网络

在开始编写实现代码之前，首先创建一个 Python 文件 main.py，然后导入以下核心模块。

```
import os
import pickle
import random
import time

import PIL
import numpy as np
import pandas as pd
import tensorflow as tf
from PIL import Image
from keras import Input, Model
from keras import backend as K
from keras.callbacks import TensorBoard
from keras.layers import Dense, LeakyReLU, BatchNormalization, ReLU, \
    Reshape, UpSampling2D, Conv2D, Activation, concatenate, Flatten, Lambda, Concatenate
from keras.optimizers import Adam
from keras_preprocessing.image import ImageDataGenerator
from matplotlib import pyplot as plt
```

1. 文本编码网络

文本编码网络仅负责将文本描述（t）转换为文本嵌入（ϕ_t）。该网络将一个句子编码为一个 1024 维的文本嵌入。前面下载好了预训练的 char-CNN-RNN 文本嵌入，下面使用该嵌入训练网络。

2. CA 网络

CA 网络负责将文本嵌入向量（ϕ_t）转换为条件潜在变量（$\hat{c}_0$）。CA 网络传递文本嵌入向量经过一个非线性全连接层，生成平均值 $\mu(\phi_t)$ 和对角协方差矩阵 $\Sigma(\phi_t)$。

CA 网络的创建方法如以下代码所示。

(1) 首先创建一个具有 256 个节点的全连接层，使用 LeakyReLU 作为激活函数。

```
input_layer = Input(shape=(1024,))
x = Dense(256)(input_layer)
mean_logsigma = LeakyReLU(alpha=0.2)(x)
```

输入形状为`(batch_size, 1024)`，输出形状为`(batch_size, 256)`。

6

(2) 然后将 mean_logsigma 拆分为 mean 和 log_sigma 两个张量。

```
mean = x[:, :128]
log_sigma = x[:, 128:]
```

该操作会创建两个张量，维度分别为(batch_size, 128)和(batch_size, 128)。

(3) 接着使用如下代码计算文本条件变量。关于如何生成文本条件变量，请参考 6.2.2 节。

```
stddev = K.exp(log_sigma)
epsilon = K.random_normal(shape=K.constant((mean.shape[1], ), dtype='int32'))
c = stddev * epsilon + mean
```

上述代码生成了维度为(batch_size, 128)的张量，即文本条件变量。CA 网络的完整代码如下。

```
def generate_c(x):
    mean = x[:, :128]
    log_sigma = x[:, 128:]

    stddev = K.exp(log_sigma)
    epsilon = K.random_normal(shape=K.constant((mean.shape[1], ), dtype='int32'))
    c = stddev * epsilon + mean

    return c
```

条件块的完整代码如下。

```
def build_ca_model():
    input_layer = Input(shape=(1024,))
    x = Dense(256)(input_layer)
    mean_logsigma = LeakyReLU(alpha=0.2)(x)

    c = Lambda(generate_c)(mean_logsigma)
    return Model(inputs=[input_layer], outputs=[c])
```

上述代码中的 build_ca_model()函数创建了一个 Keras 模型，包含一个全连接层，使用 LeakyReLU 作为激活函数。

3. 生成网络

需要创建的生成网络是一个 cGAN，以文本条件变量为条件。该生成网络接收从潜在空间采样的随机噪声向量，生成形状为 64×64×3 的图像。

下面编写生成网络的代码。

(1) 首先创建一个输入层，向网络传递输入（噪声变量）。

```
input_layer2 = Input(shape=(100, ))
```

(2) 然后将文本条件变量和噪声变量沿着维度 1 进行拼接。

```
gen_input = Concatenate(axis=1)([c, input_layer2])
```

其中，c 是文本条件变量。上一步编写了生成文本条件变量的代码，`gen_input` 将会是生成网络的输入。

(3) 接着创建一个具有 16 384（128×8×4×4）个节点的全连接层，使用 ReLU 作为激活函数，如下所示。

```
x = Dense(128 * 8 * 4 * 4, use_bias=False)(gen_input)
x = ReLU()(x)
```

(4) 然后改变上一层输出的形状，使其变成维度为`(batch_size, 4, 4,128*8)`的张量。

```
x = Reshape((4, 4, 128 * 8), input_shape=(128 * 8 * 4 * 4,))(x)
```

该操作将二维张量转换为了四维张量。

(5) 接着创建一个 2D 上采样卷积块。该块包括一个上采样层、一个卷积层，以及一个批归一化层。在进行批归一化之后，使用 ReLU 作为该块的激活函数。

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(512, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

(6) 再创建 3 个 2D 上采样卷积块，如下所示。

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(256, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(128, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(64, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

(7) 然后创建一个卷积层，生成一个低分辨率图像。

```
x = Conv2D(3, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = Activation(activation='tanh')(x)
```

(8) 最后创建一个 Keras 模型，并指明网络的输入和输出，如下所示。

```
stage1_gen = Model(inputs=[input_layer, input_layer2],
                   outputs=[x, mean_logsigma])
```

其中，x 是模型的输出，其形状为`(batch_size, 64, 64, 3)`。

生成网络的完整代码如下。

```
def build_stage1_generator():
    """
    构建一个生成网络模型
    """

    input_layer = Input(shape=(1024,))
    x = Dense(256)(input_layer)
    mean_logsigma = LeakyReLU(alpha=0.2)(x)

    c = Lambda(generate_c)(mean_logsigma)

    input_layer2 = Input(shape=(100,))

    gen_input = Concatenate(axis=1)([c, input_layer2])

    x = Dense(128 * 8 * 4 * 4, use_bias=False)(gen_input)
    x = ReLU()(x)

    x = Reshape((4, 4, 128 * 8), input_shape=(128 * 8 * 4 * 4,))(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(512, kernel_size=3, padding="same", strides=1,
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(256, kernel_size=3, padding="same", strides=1,
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(128, kernel_size=3, padding="same", strides=1,
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(64, kernel_size=3, padding="same", strides=1,
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
```

```
    x = Conv2D(3, kernel_size=3, padding="same", strides=1,
               use_bias=False)(x)
    x = Activation(activation='tanh')(x)

    stage1_gen = Model(inputs=[input_layer, input_layer2],
                       outputs=[x, mean_logsigma])
    return stage1_gen
```

该模型将 CA 网络和生成网络组合在一个网络中。它接收两个输入，返回两个输出。输入是文本嵌入和噪声变量，输出是生成的图像和 mean_logsigma。

这样就实现了生成网络，下面实现判别网络。

4. 判别网络

判别网络是分类器网络，包含一系列下采样层，将给定图像分类为真的或假的。

下面编写判别网络的代码。

(1) 首先创建一个输入层，向网络传递输入。

```
input_layer = Input(shape=(64, 64, 3))
```

(2) 然后添加一个 2D 卷积层，参数如下。

- 过滤器数量：64
- 卷积核大小：(4, 4)
- 步长：2
- 填充方式：same
- 是否使用偏置量：否
- 激活函数：LeakyReLU，alpha 值为 0.2

```
stage1_dis = Conv2D(64, (4, 4),
                    padding='same', strides=2,
                    input_shape=(64, 64, 3), use_bias=False)(input_layer)
stage1_dis = LeakyReLU(alpha=0.2)(stage1_dis)
```

(3) 接着添加两个卷积层，每个卷积层后面是一个批归一化层和一个 LeakyReLU 激活函数，参数如下。

- 过滤器数量：128
- 卷积核大小：(4, 4)
- 步长：2
- 填充方式：same
- 是否使用偏置量：否
- 激活函数：LeakyReLU，alpha 值为 0.2

```
x = Conv2D(128, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
```

(4) 再添加一个 2D 卷积层、一个批归一化层和一个 LeakyReLU 激活函数，参数如下。

- 过滤器数量：256
- 卷积核大小：(4, 4)
- 步长：2
- 填充方式：same
- 是否使用偏置量：否
- 激活函数：LeakyReLU，alpha 值为 0.2

```
x = Conv2D(256, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
```

(5) 再添加一个 2D 卷积层、一个批归一化层和一个 LeakyReLU 激活函数，参数如下。

- 过滤器数量：512
- 卷积核大小：(4, 4)
- 步长：2
- 填充方式：same
- 是否使用偏置量：否
- 激活函数：LeakyReLU，alpha 值为 0.2

```
x = Conv2D(512, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)
```

(6) 再创建一个输入层，用于接收经过空间复制和压缩的文本嵌入。

```
input_layer2 = Input(shape=(4, 4, 128))
```

(7) 添加一个拼接层，将 *x* 和 `input_layer2` 拼接起来。

```
merged_input = concatenate([x, input_layer2])
```

(8) 再添加一个 2D 卷积层、一个批归一化层和一个 LeakyReLU 激活函数，参数如下。

- 过滤器数量：512
- 卷积核大小：1
- 步长：1
- 填充方式：same
- 是否使用批归一化：是

❑ 激活函数：LeakyReLU，alpha 值为 0.2

```
x2 = Conv2D(64 * 8, kernel_size=1, padding="same", strides=1)(merged_input)
x2 = BatchNormalization()(x2)
x2 = LeakyReLU(alpha=0.2)(x2)
```

(9) 将该张量扁平化，然后添加一个全连接分类层。

```
# 将张量扁平化
x2 = Flatten()(x2)

# 分类层
x2 = Dense(1)(x2)
x2 = Activation('sigmoid')(x2)
```

(10) 最后，创建 Keras 模型。

```
stage1_dis = Model(inputs=[input_layer, input_layer2], outputs=[x2])
```

模型输出的是输入图像属于真实类别或虚假类别的概率。判别网络的完整代码如下。

```
def build_stage1_discriminator():
    input_layer = Input(shape=(64, 64, 3))

    x = Conv2D(64, (4, 4), padding='same', strides=2, input_shape=(64, 64, 3),
               use_bias=False)(input_layer)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(128, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(256, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(512, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    input_layer2 = Input(shape=(4, 4, 128))

    merged_input = concatenate([x, input_layer2])

    x2 = Conv2D(64 * 8, kernel_size=1, padding="same", strides=1)(merged_input)
    x2 = BatchNormalization()(x2)
    x2 = LeakyReLU(alpha=0.2)(x2)
    x2 = Flatten()(x2)
    x2 = Dense(1)(x2)
    x2 = Activation('sigmoid')(x2)

stage1_dis = Model(inputs=[input_layer, input_layer2], outputs=[x2])
return stage1_dis
```

该模型以低分辨率图像和压缩后的文本嵌入为输入，输出概率。至此，实现了判别网络，下面创建对抗模型。

5. 对抗模型

创建对抗模型，需要将生成网络和判别网络整合在一起，创建新的 Keras 模型。

(1) 首先创建 3 个输入层，向网络传递输入。

```
def build_adversarial_model(gen_model, dis_model):
    input_layer = Input(shape=(1024,))
    input_layer2 = Input(shape=(100,))
    input_layer3 = Input(shape=(4, 4, 128))
```

(2) 然后使用生成网络输出低分辨率图像。

```
# 获取生成网络模型的输出
x, mean_logsigma = gen_model([input_layer, input_layer2])

# 将判别网络设置为不可训练
dis_model.trainable = False
```

(3) 接着使用判别网络得出概率。

```
# 获取判别网络模型的输出
valid = dis_model([x, input_layer3])
```

(4) 最后，创建对抗模型，接收 3 个输入，返回两个输出。

```
model = Model(inputs=[input_layer, input_layer2, input_layer3],
              outputs=[valid, mean_logsigma])
return model
```

至此，对抗模型就准备好了。对抗模型可以进行端到端的训练。前面介绍了 StackGAN 第一阶段所涉及的各个网络，下面着手实现 StackGAN 第二阶段涉及的各个网络。

6.5.2 第二阶段

StackGAN 的第二阶段和第一阶段稍有不同。生成网络模型的输入是条件变量（ $\hat{c}_0$ ）和第一阶段生成网络生成的低分辨率图像。

StackGAN 第二阶段包含 5 个组件。

- 文本编码网络
- CA 网络
- 下采样块
- 残差块
- 上采样块

文本编码网络和 CA 网络与前面第一阶段使用的类似。下面介绍生成网络的 3 个组件，即下采样块、残差块和上采样块。

1. 生成网络

生成网络由 3 个模块组成。下面依次为每个模块编写代码。从下采样块开始。

- 下采样块

下采样块接收第一阶段生成网络输出的维度为 64×64×3 的低分辨率图像，对其进行下采样，然后生成形状为 16×16×512 的张量。其间图像会经过一系列 2D 卷积块。

下采样块的实现代码如下。

(1) 首先创建第 1 个下采样块。该块包含一个使用 ReLU 作为激活函数的 2D 卷积层。在执行 2D 卷积操作之前，首先对输入的四周进行零填充。该块中各层的配置如下。

- 填充大小：(1, 1)
- 过滤器数量：128
- 卷积核大小：(3, 3)
- 步长：1
- 激活函数：ReLU

```
x = ZeroPadding2D(padding=(1, 1))(input_lr_images)
x = Conv2D(128, kernel_size=(3, 3), strides=1, use_bias=False)(x)
x = ReLU()(x)
```

(2) 然后添加第 2 个卷积块，配置如下。

- 填充大小：(1, 1)
- 过滤器数量：256
- 卷积核大小：(4, 4)
- 步长：2
- 是否使用批归一化：是
- 激活函数：ReLU

```
x = ZeroPadding2D(padding=(1, 1))(x)
x = Conv2D(256, kernel_size=(4, 4), strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

(3) 再添加一个卷积块，配置如下。

- 填充大小：(1, 1)
- 过滤器数量：512
- 卷积核大小：(4, 4)

- 步长：2
- 是否使用批归一化：是
- 激活函数：ReLU

```
x = ZeroPadding2D(padding=(1, 1))(x)
x = Conv2D(512, kernel_size=(4, 4), strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

下采样块生成形状为 16×16×512 的张量。之后是一系列残差块。在将该张量传递给残差块之前，需要将它和文本条件变量拼接在一起。实现代码如下。

```
# 该块会扩展文本条件变量，然后将其和编码后的图像张量拼接在一起
def joint_block(inputs):
    c = inputs[0]
    x = inputs[1]

    c = K.expand_dims(c, axis=1)
    c = K.expand_dims(c, axis=1)
    c = K.tile(c, [1, 16, 16, 1])
    return K.concatenate([c, x], axis=3)
# 这是需要添加到生成网络的 lambda 层
c_code = Lambda(joint_block)([c, x])
```

其中，`c` 的形状是 `(batch_size, 228)`，`x` 的形状是 `(batch_size, 16, 16, 512)`。`c_code` 的形状会是 `(batch_size, 640)`。

- **残差块**

残差块包含两个 2D 卷积层，每个后面都是一个批归一化层和一个激活函数。

(1) 首先定义残差块，代码如下。

```
def residual_block(input):
    """
    生成网络中的残差块
    """
    x = Conv2D(128 * 4, kernel_size=(3, 3), padding='same',
               strides=1)(input)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(128 * 4, kernel_size=(3, 3), strides=1,
               padding='same')(x)
    x = BatchNormalization()(x)

    x = add([x, input])
    x = ReLU()(x)

    return x
```

将原始输入和第二个 2D 卷积层的输出相加，所得张量就是残差块的输出。

(2) 然后添加一个 2D 卷积块，其超参数如下。

- **填充大小**：(1, 1)
- **过滤器数量**：512
- **卷积核大小**：(3, 3)
- **步长**：1
- **是否使用批归一化**：是
- **激活函数**：ReLU

```
x = ZeroPadding2D(padding=(1, 1))(c_code)
x = Conv2D(512, kernel_size=(3, 3), strides=1, use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

(3) 接着依次添加 4 个残差块。

```
x = residual_block(x)
x = residual_block(x)
x = residual_block(x)
x = residual_block(x)
```

上采样块会接收残差块的输出向量。下面编写上采样块的代码。

- **上采样块**

上采样块包含能提高图像空间分辨率的层，生成维度为 256×256×3 的高分辨率图像。

下面编写上采样块的代码。

(1) 首先添加一个上采样块，其中包含一个 2D 上采样层、一个 2D 卷积层、一个批归一化层和一个激活函数。该块中用到的不同参数如下所示。

- **上采样大小**：(2, 2)
- **过滤器数量**：512
- **卷积核大小**：3
- **填充方式**：same
- **步长**：1
- **是否使用批归一化**：是
- **激活函数**：ReLU

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(512, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

6

(2) 然后再添加 3 个上采样块。这些上采样块所用的超参数见如下代码。

```
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(256, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(128, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(64, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)
```

(3) 最后添加卷积层。这一层负责生成高分辨率图像。

```
x = Conv2D(3, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = Activation('tanh')(x)
```

使用上面各个组件创建生成网络模型。

```
model = Model(inputs=[input_layer, input_lr_images], outputs=[x, mean_logsigma])
```

这样就准备好了生成网络模型，可用于生成高分辨率图像。清晰起见，生成网络的完整代码如下。

```
def build_stage2_generator():
    """
    创建 StackGAN 第二阶段的生成网络
    """

    # 1. CA 网络
    input_layer = Input(shape=(1024,))
    input_lr_images = Input(shape=(64, 64, 3))

    ca = Dense(256)(input_layer)
    mean_logsigma = LeakyReLU(alpha=0.2)(ca)
    c = Lambda(generate_c)(mean_logsigma)

    # 2. 图像编码网络
    x = ZeroPadding2D(padding=(1, 1))(input_lr_images)
    x = Conv2D(128, kernel_size=(3, 3), strides=1, use_bias=False)(x)
    x = ReLU()(x)

    x = ZeroPadding2D(padding=(1, 1))(x)
    x = Conv2D(256, kernel_size=(4, 4), strides=2, use_bias=False)(x)
```

```
x = BatchNormalization()(x)
x = ReLU()(x)

x = ZeroPadding2D(padding=(1, 1))(x)
x = Conv2D(512, kernel_size=(4, 4), strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

# 拼接块
c_code = Lambda(joint_block)([c, x])
# 3. 残差块
x = ZeroPadding2D(padding=(1, 1))(c_code)
x = Conv2D(512, kernel_size=(3, 3), strides=1, use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = residual_block(x)
x = residual_block(x)
x = residual_block(x)
x = residual_block(x)

# 4. 上采样块
x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(512, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(256, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(128, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = UpSampling2D(size=(2, 2))(x)
x = Conv2D(64, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = ReLU()(x)

x = Conv2D(3, kernel_size=3, padding="same", strides=1,
           use_bias=False)(x)
x = Activation('tanh')(x)

model = Model(inputs=[input_layer, input_lr_images], outputs=[x,
              mean_logsigma])
return model
```

6

2. 判别网络

StackGAN 第二阶段的判别网络包括一系列下采样层、一个拼接块，以及一个分类器。下面依次编写每个块的代码。

首先创建一个输入层，如下所示。

```
input_layer = Input(shape=(256, 256, 3))
```

- **下采样块**

下采样块的各层对图像进行下采样。

首先添加下采样块各层。代码简单易懂。

```
x = Conv2D(64, (4, 4), padding='same', strides=2, input_shape=(256, 256, 3),
           use_bias=False)(input_layer)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(128, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(256, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(512, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(1024, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(2048, (4, 4), padding='same', strides=2, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(1024, (1, 1), padding='same', strides=1, use_bias=False)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.2)(x)

x = Conv2D(512, (1, 1), padding='same', strides=1, use_bias=False)(x)
x = BatchNormalization()(x)

x2 = Conv2D(128, (1, 1), padding='same', strides=1, use_bias=False)(x)
x2 = BatchNormalization()(x2)
x2 = LeakyReLU(alpha=0.2)(x2)

x2 = Conv2D(128, (3, 3), padding='same', strides=1, use_bias=False)(x2)
x2 = BatchNormalization()(x2)
```

```
x2 = LeakyReLU(alpha=0.2)(x2)

x2 = Conv2D(512, (3, 3), padding='same', strides=1, use_bias=False)(x2)
x2 = BatchNormalization()(x2)
```

现有两个输出：`x` 和 `x2`。将这两个张量相加，创建一个形状相同的张量，并应用 LeakyReLU 激活函数。

```
added_x = add([x, x2])
added_x = LeakyReLU(alpha=0.2)(added_x)
```

- **拼接块**

创建另一个输入层，用于接收经过空间复制和压缩的嵌入。

```
input_layer2 = Input(shape=(4, 4, 128))
```

将下采样块的输出和空间压缩后的嵌入拼接在一起。

```
input_layer2 = Input(shape=(4, 4, 128))

merged_input = concatenate([added_x, input_layer2])
```

- **全连接分类器**

将合并后的输入传递给用于分类的块，该块拥有一个卷积层和一个全连接层。

```
x3 = Conv2D(64 * 8, kernel_size=1, padding="same", strides=1)(merged_input)
x3 = BatchNormalization()(x3)
x3 = LeakyReLU(alpha=0.2)(x3)
x3 = Flatten()(x3)
x3 = Dense(1)(x3)
x3 = Activation('sigmoid')(x3)
```

`x3` 是判别网络的输出，即给定图像属于真实类别或虚假类别的概率。

最后，创建一个模型。

```
stage2_dis = Model(inputs=[input_layer, input_layer2], outputs=[x3])
```

如上所示，该模型接收两个输入，返回一个输出。

判别网络的完整代码如下。

```
def build_stage2_discriminator():
    input_layer = Input(shape=(256, 256, 3))

    x = Conv2D(64, (4, 4), padding='same', strides=2, input_shape=(256, 256, 3),
               use_bias=False)(input_layer)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(128, (4, 4), padding='same', strides=2, use_bias=False)(x)
```

6

```
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(256, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(512, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(1024, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(2048, (4, 4), padding='same', strides=2, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(1024, (1, 1), padding='same', strides=1, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2D(512, (1, 1), padding='same', strides=1, use_bias=False)(x)
    x = BatchNormalization()(x)

    x2 = Conv2D(128, (1, 1), padding='same', strides=1, use_bias=False)(x)
    x2 = BatchNormalization()(x2)
    x2 = LeakyReLU(alpha=0.2)(x2)

    x2 = Conv2D(128, (3, 3), padding='same', strides=1, use_bias=False)(x2)
    x2 = BatchNormalization()(x2)
    x2 = LeakyReLU(alpha=0.2)(x2)

    x2 = Conv2D(512, (3, 3), padding='same', strides=1, use_bias=False)(x2)
    x2 = BatchNormalization()(x2)
    added_x = add([x, x2])
    added_x = LeakyReLU(alpha=0.2)(added_x)

    input_layer2 = Input(shape=(4, 4, 128))
    # 拼接块
    merged_input = concatenate([added_x, input_layer2])

    x3 = Conv2D(64 * 8, kernel_size=1, padding="same", strides=1)(merged_input)
    x3 = BatchNormalization()(x3)
    x3 = LeakyReLU(alpha=0.2)(x3)
    x3 = Flatten()(x3)
    x3 = Dense(1)(x3)
    x3 = Activation('sigmoid')(x3)

    stage2_dis = Model(inputs=[input_layer, input_layer2], outputs=[x3])
    return stage2_dis
```

这样就创建好了 StackGAN 第一阶段和第二阶段的模型。下面训练模型。

6.6 训练 StackGAN

下面介绍 StackGAN 两个阶段的训练方法。首先训练 StackGAN 的第一阶段，然后训练 StackGAN 的第二阶段。

6.6.1 训练 StackGAN 的第一阶段

开始训练之前，首先需要确定核心超参数。超参数是训练中的固定数值。下面定义超参数。

```
data_dir = "Specify your dataset directory here/Data/birds"
train_dir = data_dir + "/train"
test_dir = data_dir + "/test"
image_size = 64
batch_size = 64
z_dim = 100
stage1_generator_lr = 0.0002
stage1_discriminator_lr = 0.0002
stage1_lr_decay_step = 600
epochs = 1000
condition_dim = 128

embeddings_file_path_train = train_dir + "/char-CNN-RNN-embeddings.pickle"
embeddings_file_path_test = test_dir + "/char-CNN-RNN-embeddings.pickle"

filenames_file_path_train = train_dir + "/filenames.pickle"
filenames_file_path_test = test_dir + "/filenames.pickle"

class_info_file_path_train = train_dir + "/class_info.pickle"
class_info_file_path_test = test_dir + "/class_info.pickle"

cub_dataset_dir = data_dir + "/CUB_200_2011"
```

然后需要加载数据集。

1. 加载数据集

加载数据集分为几步，下面依次介绍。

(1) 首先需要加载存储在一个 pickle 文件中的类别 ID。以下代码加载类别 ID，返回包含所有类别 ID 的列表。

```
def load_class_ids(class_info_file_path):
    """
    从 class_info.pickle 文件中加载类别 id
    """
    with open(class_info_file_path, 'rb') as f:
        class_ids = pickle.load(f, encoding='latin1')
        return class_ids
```

(2) 然后加载（存储在 pickle 文件中的）文件名。实现代码如下。

```
def load_filenames(filenames_file_path):
    """
    加载 filenames.pickle 文件，返回包含所有文件名的列表
    """
    with open(filenames_file_path, 'rb') as f:
        filenames = pickle.load(f, encoding='latin1')
    return filenames
```

(3) 接着加载（存储在 pickle 文件中的）文本嵌入。加载文件并提取文本嵌入，如下所示。

```
def load_embeddings(embeddings_file_path):
    """
    加载嵌入
    """
    with open(embeddings_file_path, 'rb') as f:
        embeddings = pickle.load(f, encoding='latin1')
        embeddings = np.array(embeddings)
        print('embeddings: ', embeddings.shape)
    return embeddings
```

(4) 然后获取边界框，用于从原始图像中提取对象。以下代码清晰展示了实现方法。

```
def load_bounding_boxes(dataset_dir):
    """
    加载边界框，返回包含文件名和对应边界框的字典
    """
    # 路径
    bounding_boxes_path = os.path.join(dataset_dir, 'bounding_boxes.txt')
    file_paths_path = os.path.join(dataset_dir, 'images.txt')

    # 读取 bounding_boxes.txt 和 images.txt 文件
    df_bounding_boxes = pd.read_csv(bounding_boxes_path,
                                    delim_whitespace=True,
                                    header=None).astype(int)
    df_file_names = pd.read_csv(file_paths_path,
                                delim_whitespace=True, header=None)

    # 创建一个包含文件名的列表
    file_names = df_file_names[1].tolist()

    # 创建一个包含文件名和边界框的字典
    filename_boundingbox_dict = {
        img_file[:-4]: [] for img_file in file_names[:2]}

    # 为图像指定相应的边界框
    for i in range(0, len(file_names)):
        # 获取边界框
        bounding_box = df_bounding_boxes.iloc[i][1:].tolist()
        key = file_names[i][:-4]
        filename_boundingbox_dict[key] = bounding_box

    return filename_boundingbox_dict
```

(5) 接着编写一个函数来加载并裁剪图像。如下代码会加载图像并按照指定的边界框进行裁剪，并将图像缩放到特定大小。

```
def get_img(img_path, bbox, image_size):
    """
    加载图像并进行缩放
    """
    img = Image.open(img_path).convert('RGB')
    width, height = img.size
    if bbox is not None:
        R = int(np.maximum(bbox[2], bbox[3]) * 0.75)
        center_x = int((2 * bbox[0] + bbox[2]) / 2)
        center_y = int((2 * bbox[1] + bbox[3]) / 2)
        y1 = np.maximum(0, center_y - R)
        y2 = np.minimum(height, center_y + R)
        x1 = np.maximum(0, center_x - R)
        x2 = np.minimum(width, center_x + R)
        img = img.crop([x1, y1, x2, y2])
        img = img.resize(image_size, PIL.Image.BILINEAR)
    return img
```

(6) 然后将以上所有函数整合在一起，获取训练所需的数据集。如下代码会返回所有图像、对应标签，以及对应的嵌入用于训练。

```
def load_dataset(filenames_file_path, class_info_file_path,
cub_dataset_dir, embeddings_file_path, image_size):
    filenames = load_filenames(filenames_file_path)
    class_ids = load_class_ids(class_info_file_path)
    bounding_boxes = load_bounding_boxes(cub_dataset_dir)
    all_embeddings = load_embeddings(embeddings_file_path)

    X, y, embeddings = [], [], []
    # TODO: 修改文件名索引
    for index, filename in enumerate(filenames[:500]):
        # 输出(class_ids[index], filenames[index])
        bounding_box = bounding_boxes[filename]

        try:
            # 加载图像
            img_name = '{}/images/{}.jpg'.format(cub_dataset_dir,
                                                  filename)
            img = get_img(img_name, bounding_box, image_size)

            all_embeddings1 = all_embeddings[index, :, :]

            embedding_ix = random.randint(0, all_embeddings1.shape[0] - 1)
            embedding = all_embeddings1[embedding_ix, :]

            X.append(np.array(img))
            y.append(class_ids[index])
            embeddings.append(embedding)
        except Exception as e:
```

```
            print(e)

    X = np.array(X)
    y = np.array(y)
    embeddings = np.array(embeddings)

    return X, y, embeddings
```

(7) 最后，在训练中加载数据集。

```
X_train, y_train, embeddings_train = load_dataset(
    filenames_file_path=filenames_file_path_train,
    class_info_file_path=class_info_file_path_train,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_train,
    image_size=(64, 64))

X_test, y_test, embeddings_test = load_dataset(
    filenames_file_path=filenames_file_path_test,
    class_info_file_path=class_info_file_path_test,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_test,
    image_size=(64, 64))
```

这样就加载好了训练所需的数据集，下面创建所需模型。

2. 创建模型

下面使用 6.5.1 节的函数创建模型。需要用到 4 个模型：1 个生成网络模型、1 个判别网络模型、1 个用于压缩文本嵌入的压缩网络模型，以及 1 个包含生成网络和判别网络的对抗模型。

(1) 首先定义训练所需的优化器。

```
dis_optimizer = Adam(lr=stage1_discriminator_lr, beta_1=0.5,
                     beta_2=0.999)
gen_optimizer = Adam(lr=stage1_generator_lr, beta_1=0.5,
                     beta_2=0.999)
```

(2) 然后构建并编译各个网络，如下所示。

```
ca_model = build_ca_model()
ca_model.compile(loss="binary_crossentropy", optimizer="adam")

stage1_dis = build_stage1_discriminator()
stage1_dis.compile(loss='binary_crossentropy',
                   optimizer=dis_optimizer)

stage1_gen = build_stage1_generator()
stage1_gen.compile(loss="mse", optimizer=gen_optimizer)

embedding_compressor_model = build_embedding_compressor_model()
embedding_compressor_model.compile(loss="binary_crossentropy",
                                   optimizer="adam")
```

```
adversarial_model = build_adversarial_model(gen_model=stage1_gen,
                                            dis_model=stage1_dis)
adversarial_model.compile(loss=['binary_crossentropy', KL_loss],
                          loss_weights=[1, 2.0], optimizer=gen_optimizer,
                          metrics=None)
```

其中，`KL_loss` 是自定义的损失函数，定义如下。

```
def KL_loss(y_true, y_pred):
    mean = y_pred[:, :128]
    logsigma = y_pred[:, :128]
    loss = -logsigma + .5 * (-1 + K.exp(2. * logsigma) + K.square(mean))
    loss = K.mean(loss)
    return loss
```

准备好了数据集和模型，下面开始训练模型。

使用 TensorBoard 存储损失，以进行可视化，如下所示。

```
tensorboard = TensorBoard(log_dir="logs/".format(time.time()))
tensorboard.set_model(stage1_gen)
tensorboard.set_model(stage1_dis)
tensorboard.set_model(ca_model)
tensorboard.set_model(embedding_compressor_model)
```

3. 训练模型

训练模型分为以下几步。

(1) 分别创建包含真实标签和虚假标签的两个张量，在训练生成网络和判别网络时候会用到它们。使用第 1 章介绍过的标签平滑技术。

```
real_labels = np.ones((batch_size, 1), dtype=float) * 0.9
fake_labels = np.zeros((batch_size, 1), dtype=float) * 0.1
```


(2) 然后创建一个 `for` 循环，运行次数和指定训练轮数相同，如下所示。

```
for epoch in range(epochs):
    print("========================================")
    print("Epoch is:", epoch)
    print("Number of batches", int(X_train.shape[0] / batch_size))

    gen_losses = []
    dis_losses = []
```

(3) 接着计算批次数量，并且编写一个 `for` 循环，运行次数和指定批次数量相同。

```
number_of_batches = int(X_train.shape[0] / batch_size)
for index in range(number_of_batches):
    print("Batch:{}".format(index+1))
```

(4) 在当前迭代中，采样一批次数据（一小批次）。创建一个噪声向量，选择一批次图像和一

批次嵌入，然后将图像归一化。

```
# 创建一批次噪声向量
z_noise = np.random.normal(0, 1, size=(batch_size, z_dim))
image_batch = X_train[index * batch_size:(index + 1) * batch_size]
embedding_batch = embeddings_train[index * batch_size:(index + 1) * batch_size]

# 将图像归一化
image_batch = (image_batch - 127.5) / 127.5
```

(5) 然后向生成网络模型传递 `embedding_batch` 和 `z_noize`，以生成假图像。

`embedding_batch` 和 `z_noise`：

```
fake_images, _ = stage1_gen.predict([embedding_batch, z_noise], verbose=3)
```

这样会生成以一批次嵌入和一批次噪声向量为条件的一批次假图像。

(6) 使用压缩网络模型压缩嵌入，然后对其进行空间复制，转换成形状为`(batch_size, 4, 4, 128)`的向量。

```
compressed_embedding = embedding_compressor_model.predict_on_batch(embedding_batch)
compressed_embedding = np.reshape(compressed_embedding,
                                  (-1, 1, 1, condition_dim))
compressed_embedding = np.tile(compressed_embedding, (1, 4, 4, 1))
```

(7) 接着使用生成网络生成的假图像、真实数据集中的真实图像，以及错误图像来训练判别网络模型。

```
dis_loss_real = stage1_dis.train_on_batch([image_batch, compressed_embedding],
                                          np.reshape(real_labels,
                                                     (batch_size, 1)))
dis_loss_fake = stage1_dis.train_on_batch([fake_images, compressed_embedding],
                                          np.reshape(fake_labels,
                                                     (batch_size, 1)))
dis_loss_wrong = stage1_dis.train_on_batch([image_batch[:(batch_size - 1)],
                                            compressed_embedding[1:]],
                                           np.reshape(fake_labels[1:],
                                                      (batch_size-1, 1)))
```

至此，判别网络就完成了在真实图像、虚假图像和错误图像 3 组数据上的训练。下面训练对抗模型。

(8) 向对抗模型提供 3 个输入以及对应的真实值。该操作计算一批次数据的梯度，并且更新其权重。

```
g_loss = adversarial_model.train_on_batch([embedding_batch,
                                            z_noise, compressed_embedding],
                                           [K.ones((batch_size, 1)) * 0.9,
                                            K.ones((batch_size, 256)) * 0.9])
```

(9) 然后计算损失并存储，以用于评估。持续输出不同的损失有助于掌握训练进程。

```
d_loss = 0.5 * np.add(dis_loss_real, 0.5 * np.add(dis_loss_wrong,
                      dis_loss_fake))

        print("d_loss:{}".format(d_loss))
        print("g_loss:{}".format(g_loss))

        dis_losses.append(d_loss)
        gen_losses.append(g_loss)
```

(10) 在完成每轮训练后，将所有损失写入 TensorBoard 中。

```
write_log(tensorboard, 'discriminator_loss', np.mean(dis_losses), epoch)
write_log(tensorboard, 'generator_loss', np.mean(gen_losses[0]), epoch)
```

(11) 在每轮训练后，生成图像并保存到结果目录中，以评估训练进程。

```
z_noise2 = np.random.normal(0, 1, size=(batch_size, z_dim))
embedding_batch = embeddings_test[0:batch_size]
fake_images, _ = stage1_gen.predict_on_batch([embedding_batch, z_noise2])

# 保存图像
for i, img in enumerate(fake_images[:10]):
    save_rgb_img(img, "results/gen_{}_{}.png".format(epoch, i))
```

其中，`save_rgb_img()`是一个效用函数，其定义如下。

```
def save_rgb_img(img, path):
    """
    保存 rgb 图像
    """
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.imshow(img)
    ax.axis("off")
    ax.set_title("Image")

    plt.savefig(path)
    plt.close()
```

(12) 保存 StackGAN 第一阶段中每个模型的权重。

```
stage1_gen.save_weights("stage1_gen.h5")
stage1_dis.save_weights("stage1_dis.h5")
```

祝贺，已经顺利完成了 StackGAN 第一阶段的训练。训练好的生成网络可以生成维度为 64×64×3 的图像。这些图像具有基础的颜色和形状。下面训练 StackGAN 的第二阶段。

6

6.6.2　训练 StackGAN 的第二阶段

训练 StackGAN 第二阶段的步骤如下。

首先指定 StackGAN 第二阶段训练所使用的超参数。

```
# 指定超参数
data_dir = "Path to the dataset/Data/birds"
train_dir = data_dir + "/train"
test_dir = data_dir + "/test"
hr_image_size = (256, 256)
lr_image_size = (64, 64)
batch_size = 8
z_dim = 100
stage1_generator_lr = 0.0002
stage1_discriminator_lr = 0.0002
stage1_lr_decay_step = 600
epochs = 10
condition_dim = 128

embeddings_file_path_train = train_dir + "/char-CNN-RNN-embeddings.pickle"
embeddings_file_path_test = test_dir + "/char-CNN-RNN-embeddings.pickle"

filenames_file_path_train = train_dir + "/filenames.pickle"
filenames_file_path_test = test_dir + "/filenames.pickle"

class_info_file_path_train = train_dir + "/class_info.pickle"
class_info_file_path_test = test_dir + "/class_info.pickle"

cub_dataset_dir = data_dir + "/CUB_200_2011"
```

1. 加载数据集

使用之前定义的函数，分别加载低分辨率数据集和高分辨率数据集。训练集和测试集需要单独加载。

```
X_hr_train, y_hr_train, embeddings_train = load_dataset(
    filenames_file_path=filenames_file_path_train,
    class_info_file_path=class_info_file_path_train,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_train, image_size=(256, 256))

X_hr_test, y_hr_test, embeddings_test = load_dataset(
    filenames_file_path=filenames_file_path_test,
    class_info_file_path=class_info_file_path_test,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_test, image_size=(256, 256))

X_lr_train, y_lr_train, _ = load_dataset(
    filenames_file_path=filenames_file_path_train,
    class_info_file_path=class_info_file_path_train,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_train, image_size=(64, 64))
```

```
X_lr_test, y_lr_test, _ = load_dataset(
    filenames_file_path=filenames_file_path_test,
    class_info_file_path=class_info_file_path_test,
    cub_dataset_dir=cub_dataset_dir,
    embeddings_file_path=embeddings_file_path_test, image_size=(64, 64))
```

2. 创建模型

如 6.5.1 节所示，创建 Keras 模型。

首先定义训练所需的优化器。

```
dis_optimizer = Adam(lr=stage1_discriminator_lr, beta_1=0.5, beta_2=0.999)
gen_optimizer = Adam(lr=stage1_generator_lr, beta_1=0.5, beta_2=0.999)
```

选择 Adam 优化器，设置学习速率为 0.0002、`beta_1` 为 0.5、`beta_2` 为 0.999。

下面构建并编译模型。

```
stage2_dis = build_stage2_discriminator()
stage2_dis.compile(loss='binary_crossentropy', optimizer=dis_optimizer)

stage1_gen = build_stage1_generator()
stage1_gen.compile(loss="binary_crossentropy", optimizer=gen_optimizer)

stage1_gen.load_weights("stage1_gen.h5")

stage2_gen = build_stage2_generator()
stage2_gen.compile(loss="binary_crossentropy", optimizer=gen_optimizer)

embedding_compressor_model = build_embedding_compressor_model()
embedding_compressor_model.compile(loss='binary_crossentropy',
                                   optimizer='adam')

adversarial_model = build_adversarial_model(stage2_gen, stage2_dis, stage1_gen)
adversarial_model.compile(loss=['binary_crossentropy', KL_loss],
                          loss_weights=[1.0, 2.0],
                          optimizer=gen_optimizer, metrics=None)
```

KL_loss 是一个自定义损失函数，前面已经定义过了。

至此，StackGAN 第二阶段所需的数据集和模型就准备好了，下面训练模型。

3. 训练模型

训练模型的步骤如下。

(1) 首先添加 TensorBoard 以存储损失。

```
tensorboard = TensorBoard(log_dir="logs/".format(time.time()))
tensorboard.set_model(stage2_gen)
tensorboard.set_model(stage2_dis)
```

(2) 然后分别创建真实类别标签和虚假类别标签的张量，在训练生成网络和判别网络时需会用到它们。运用第 1 章介绍过的标签平滑技术。

```
real_labels = np.ones((batch_size, 1), dtype=float) * 0.9
fake_labels = np.zeros((batch_size, 1), dtype=float) * 0.1
```

(3) 接着创建一个 `for` 循环，运行次数和指定训练轮数相同，如下所示。

```
for epoch in range(epochs):
    print("========================================")
    print("Epoch is:", epoch)

    gen_losses = []
    dis_losses = []
```

(4) 在训练轮循环内部再创建一个循环，运行次数和指定批次数相同。

```
print("Number of batches:{}".format(number_of_batches))
for index in range(number_of_batches):
    print("Batch:{}".format(index))
```

(5) 采样训练所需的数据。

```
# 创建一小批次噪声向量
z_noise = np.random.normal(0, 1, size=(batch_size, z_dim))
X_hr_train_batch = X_hr_train[index * batch_size:(index + 1) * batch_size]
embedding_batch = embeddings_train[index * batch_size:(index + 1) * batch_size]

# 将图像归一化
X_hr_train_batch = (X_hr_train_batch - 127.5) / 127.5
```

(6) 接着使用生成网络生成维度为 256×256×2 的假图像。在这一步中，首先使用第一阶段生成网络生成低分辨率的假图像，然后使用第二阶段生成网络基于低分辨率图像生成高分辨率图像。

```
lr_fake_images, _ = stage1_gen.predict([embedding_batch, z_noise],
                                        verbose=3)
hr_fake_images, _ = stage2_gen.predict([embedding_batch, lr_fake_images],
                                        verbose=3)
```

(7) 使用压缩网络模型对嵌入进行压缩和空间复制，将其转换成形状为 `(batch_size, 4, 4, 128)` 的张量。

```
compressed_embedding = embedding_compressor_model.predict_on_batch(embedding_batch)
compressed_embedding = np.reshape(compressed_embedding,
                                  (-1, 1, 1, condition_dim))
compressed_embedding = np.tile(compressed_embedding, (1, 4, 4, 1))
```

(8) 接着使用假图像、真图像和错误图像训练判别网络。

```
dis_loss_real = stage2_dis.train_on_batch([X_hr_train_batch, compressed_embedding],
                                          np.reshape(real_labels,
                                                     (batch_size, 1)))
dis_loss_fake = stage2_dis.train_on_batch([hr_fake_images,
```

```
                                             compressed_embedding],
                                            np.reshape(fake_labels,
                                             (batch_size, 1)))
dis_loss_wrong = stage2_dis.train_on_batch([X_hr_train_batch[:(batch_size - 1)],
                                             compressed_embedding[1:]],
                                             np.reshape(fake_labels[1:],
                                                        (batch_size-1, 1)))
```

(9) 然后训练对抗模型，这是生成网络模型和判别网络模型的组合。向对抗模型提供 3 个输入和对应的真实值。

```
g_loss = adversarial_model.train_on_batch([embedding_batch,
                                             z_noise, compressed_embedding],
                                            [K.ones((batch_size, 1)) * 0.9,
                                             K.ones((batch_size, 256)) * 0.9])
```

(10) 计算损失并存储，以用于评估。

```
d_loss = 0.5 * np.add(dis_loss_real, 0.5 * np.add(dis_loss_wrong, dis_loss_fake))
print("d_loss:{}".format(d_loss))

print("g_loss:{}".format(g_loss))
```

每轮训练后，将损失写入 TensorBoard 中。

```
write_log(tensorboard, 'discriminator_loss',
          np.mean(dis_losses), epoch)
write_log(tensorboard, 'generator_loss',
          np.mean(gen_losses)[0], epoch)
```

(11) 每轮训练过后，生成图像并保存到结果目录，以评估训练进程。以下代码只保存生成的第一张图像，可适当修改，以保存所需图像。

```
# 每两轮生成并保存图像
if epoch % 2 == 0:
    # z_noise2 = np.random.uniform(-1, 1, size=(batch_size, z_dim))
    z_noise2 = np.random.normal(0, 1, size=(batch_size, z_dim))
    embedding_batch = embeddings_test[0:batch_size]

    lr_fake_images, _ = stage1_gen.predict([embedding_batch, z_noise2],
                                            verbose=3)
    hr_fake_images, _ = stage2_gen.predict([embedding_batch, lr_fake_images],
                                            verbose=3)

# 保存图像
for i, img in enumerate(hr_fake_images[:10]):
    save_rgb_img(img, "results2/gen_{}_{}.png".format(epoch, i))
```

其中，save_rgb_img()是一个效用函数，是在 6.6.1 节定义的。

(12) 最后，保存模型权重值。

6

```
# 保存模型
stage2_gen.save_weights("stage2_gen.h5")
stage2_dis.save_weights("stage2_dis.h5")
```

祝贺，顺利完成了 StackGAN 第二阶段的训练。所得生成网络可以生成维度为 256×256×3 的逼真图像。如果向该生成网络提供文本嵌入和噪声向量，可以生成分辨率为 256×256×3 的图像。下面将网络的损失图可视化。

6.6.3　生成图像可视化

500 轮训练后，生成网络会开始生成比较不错的图像，如图 6-7 所示。



图 6-7　StackGAN 第一阶段和第二阶段生成的图像

建议将网络训练 1000 轮。一切顺利的话，1000 轮训练后，生成网络会开始生成逼真的图像。

6.6.4　损失可视化

启动 TensorBoard 服务器，将训练损失可视化。如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图，如下所示。

第一阶段判别网络的损失图如下（见图 6-8）。

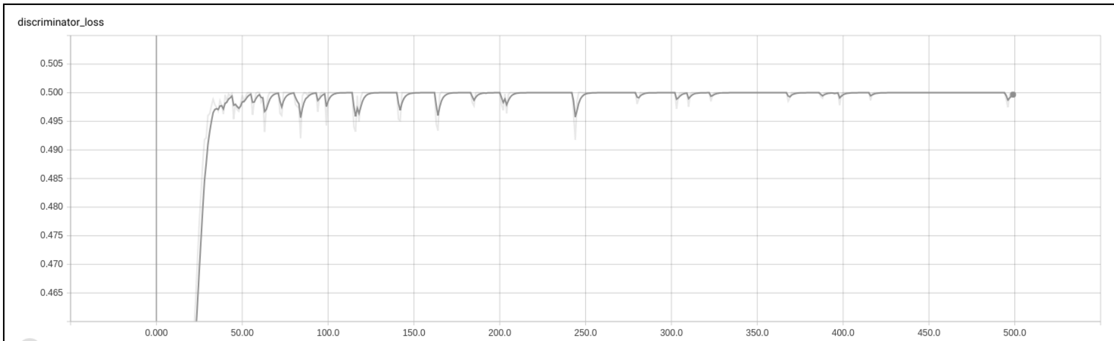


图 6-8 第一阶段判别网络的损失图

第一阶段生成网络的损失图如下（见图 6-9）。

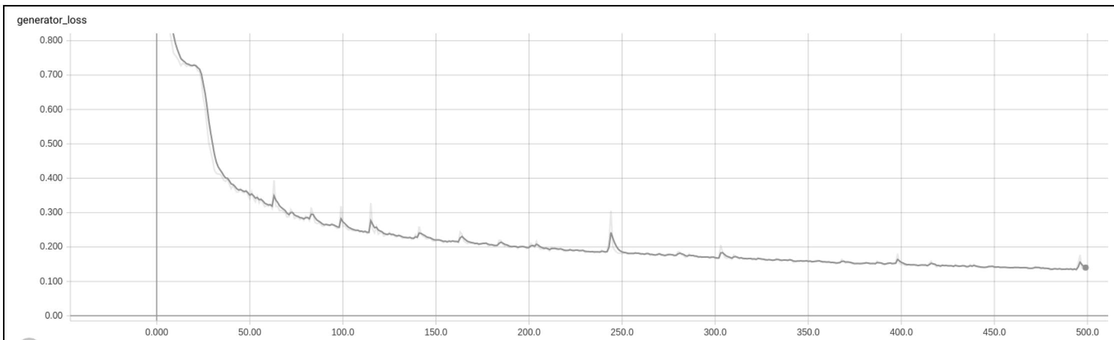


图 6-9 第一阶段生成网络的损失图

类似地，也可以通过 TensorBoard 获得第二阶段生成网络和判别网络的损失图。

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

6.6.5 图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 6-10）。

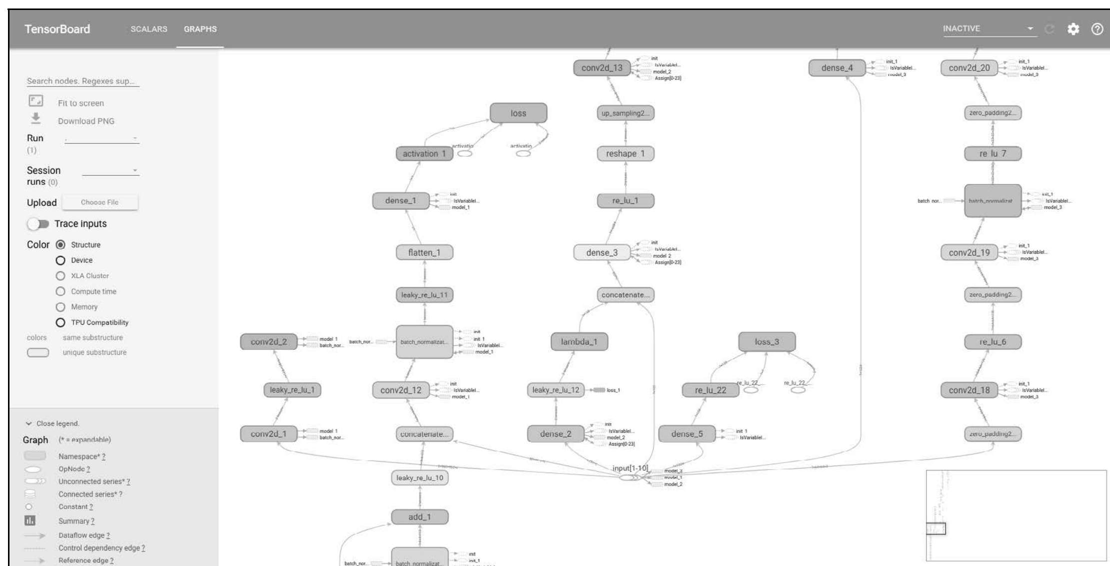


图 6-10　各图中的张量和不同运算的流

6.7　StackGAN 的实际应用

StackGAN 的行业应用包括以下几种。

- 自动生成高分辨率图像，用于娱乐或教育。
- **漫画创作**：利用 StackGAN，漫画创作过程可以缩短到几日，因为 StackGAN 可以自动生成漫画，协助创作。
- **电影创作**：StackGAN 可以基于文本生成帧，从而协助电影创作者。
- **艺术创作**：StackGAN 可以基于文本生成草图，从而协助艺术家

6.8　小结

本章介绍了基于文本生成高分辨率图像的 StackGAN 的实现方法。首先简单介绍了 StackGAN，探讨了 StackGAN 的具体架构和训练 StackGAN 所用的损失函数，然后下载并准备了数据集，接着使用 Keras 框架实现了 StackGAN，随后依次训练了 StackGAN 的第一阶段和第二阶段，最后评估了训练后的模型，保存成果以供后续使用。

下一章会介绍 CycleGAN，它可以将绘画转换成照片。

第 7 章

使用 CycleGAN 将绘画转换为照片

CycleGAN 用于跨领域变换，比如改变图像的风格、将绘画转换成照片（或者反过来）、照片增强、改变照片的季节，等等。CycleGAN 是由 Jun-Yan Zhu、Taesung Park、Phillip Isola 和 Alexei A. Efros 在论文“npaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”中提出的。该论文于 2018 年 2 月在加州大学伯克利分校的伯克利人工智能研究院（Berkeley AI Research，BAIR）完成。由于用途广泛，CycleGAN 在 GAN 社区引起了一阵轰动。本章介绍 CycleGAN，并使用 CycleGAN 将绘画转换成照片。

本章将讨论以下主题。

- CycleGAN 简介
- CycleGAN 架构
- 数据收集和准备
- CycleGAN 的 Keras 实现
- 目标函数
- 训练 CycleGAN
- CycleGAN 的实际应用

7.1 CycleGAN 简介

如果要用普通 GAN 将照片转换为绘画（或着反过来），需要使用成对的图像进行训练。而 CycleGAN 是一种特殊的 GAN，无须使用成对图像进行训练，便可以将图像从一个领域 X 变换到另一个领域 Y。CycleGAN 训练学习两种映射的生成网络。绝大多数 GAN 训练只一个生成网络，而 CycleGAN 会训练两个生成网络和两个判别网络。

CycleGAN 包含如下两个生成网络。

- **生成网络 A**：学习映射 $G: X \rightarrow Y$，其中 X 是源领域，Y 是目标领域。该映射接收源领域 A 的图像，将其转换成和目标领域 B 中的图像相似的图像。简单说来，该网络旨在学习能使 $G(X)$ 和 Y 相似的映射。
- **生成网络 B**：学习映射 $F: Y \rightarrow X$，接收目标领域 B 的图像，将其转换成和源领域 A 中图像相似的图像。类似地，该网络旨在学习能使 $F(G(X))$ 和 X 相似的映射。

两个网络的架构相同，但都单独训练。

CycleGAN 包含如下两个判别网络。

- **判别网络 A**：判别网络 A 负责区分生成网络 B 生成的图像（用 $F(Y)$ 表示）和源领域 A 中的真实图像（表示为 X）。
- **判别网络 B**：判别网络 B 负责区分生成网络 A 生成的图像（用 $G(X)$ 表示）和目标领域 B 中的真实图像（表示为 Y）。

两个网络的架构相同。类似于生成网络，两个判别网络需要单独训练，如图 7-1 所示。

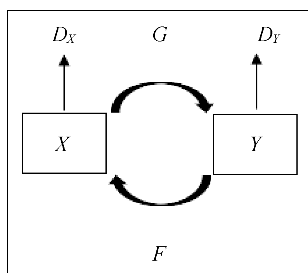


图 7-1　拥有两个生成网络和两个判别网络的对抗模型

下面具体介绍 CycleGAN 架构。

7.1.1　CycleGAN 架构

CycleGAN 由生成网络架构和判别网络架构组成。生成网络架构用于创建两个模型：生成网络 A 和生成网络 B。判别网络架构用于创建另外两个模型：判别网络 A 和判别网络 B。下面依次介绍两种网络架构。

1. 生成网络架构

该生成网络是一种自编码网络，接收图像作为输入，输出其他图像。它由一个编码网络和一个解码网络组成。编码网络包含能进行下采样的卷积层，将形状为 128×128×3 的输入转换成内部表示。解码网络包含两个上采样块和一个最终的卷积层，将内部表示转换成形状为 128×128×3 的输出。

生成网络由如下这些块组成。

- 卷积块
- 残差块
- 上采样块
- 最终的卷积层

下面逐一介绍这些块。

- **卷积块**：卷积块包含一个 2D 卷积层和一个实例归一化层，使用 ReLU 作为激活函数。关于实例归一化的更多信息，请参考第 1 章。

  生成网络包含 3 个卷积块，各卷积块的配置如表 7-1 所示。

表　7-1

层　名　称	超　参　数	输入形状	输出形状
2D 卷积层	filters=32, kernel_size=7, strides=1, padding='same'	(128, 128, 32)	(128, 128, 32)
实例归一化层	axis=1	(128, 128, 32)	(128, 128, 32)
激活层	activation='relu'	(128, 128, 32)	(128, 128, 32)
2D 卷积层	filters=64, kernel_size=3, strides=2, padding='same'	(128, 128, 32)	(64, 64, 64)
实例归一化层	axis=1	(64, 64, 64)	(64, 64, 64)
激活层	activation='relu'	(64, 64, 64)	(64, 64, 64)
2D 卷积层	filters=128, kernel_size=3, strides=2, padding='same'	(64, 64, 64)	(32, 32, 128)
实例归一化层	axis=1	(32, 32, 128)	(32, 32, 128)
激活层	activation='relu'	(32, 32, 128)	(32, 32, 128)

- **残差块**：残差块包含两个 2D 卷积层，每个卷积层后面都有一个批归一化层，其 momentum 值为 0.8。生成网络包含 6 个残差块，各残差块的配置如表 7-2 所示。

表　7-2

层　名　称	超　参　数	输入形状	输出形状
2D 卷积层	filters=128, kernel_size=3, strides=1, padding='same'	(32, 32, 128)	(32, 32, 128)
批归一化层	axis=3, momentum=0.9, epsilon=1e-5	(32, 32, 128)	(32, 32, 128)

（续）

层 名 称	超 参 数	输入形状	输出形状
2D 卷积层	filters=138, kernel_size=3, strides=1, padding='same'	(32, 32, 128)	(32, 32, 128)
批归一化层	axis=3, momentum=0.9, epsilon=1e-5	(32, 32, 128)	(32, 32, 128)
加法层	无	(32, 32, 128)	(32, 32, 128)

加法层计算该块的输入张量和最后的批归一化层的输出之和。

- **上采样块**：上采样块包含一个 2D 转置卷积层，使用 ReLU 作为激活函数。生成网络包含两个上采样块。第一个上采样块的配置如表 7-3 所示。

表 7-3

层 名 称	超 参 数	输入形状	输出形状
2D 转置卷积层	filters=64, kernel_size=3, strides=2, padding='same', use_bias=False	(32, 32, 128)	(64, 64, 64)
实例归一化层	axis=1	(64, 64, 64)	(64, 64, 64)
激活层	activation='relu'	(64, 64, 64)	(64, 64, 64)

第二个上采样块的配置如表 7-4 所示。

表 7-4

层 名 称	超 参 数	输入形状	输出形状
2D 转置卷积层	filters=32, kernel_size=3, strides=2, padding='same', use_bias=False	(64, 64, 64)	(128, 128, 32)
实例归一化层	axis=1	(128, 128, 32)	(128, 128, 32)
激活层	activation='relu'	(128, 128, 32)	(128, 128, 32)

- **最后的卷积层**：最后一层是一个 2D 卷积层，使用 tanh 作为激活函数。该层生成形状为 (256, 256, 3) 的图像。最后一层的配置如表 7-5 所示。

表 7-5

层 名 称	超 参 数	输入形状	输出形状
2D 卷积层	filters=3, kernel_size=7, strides=1, padding='same', activation='tanh'	(128, 128, 32)	(128, 128, 3)



这些超参数是针对 Keras 框架设计的。如果使用其他框架，请做相应调整。

2. 判别网络架构

判别网络的架构类似于 PatchGAN 中的判别网络架构，是一个包含几个卷积块的深度卷积神经网络。简单说来，判别网络接收形状为(128, 128, 3)的图像，然后判断该图像是真是假。判别网络包含几个 2D 零填充层。判别网络的具体架构见表 7-6。

表 7-6

层 名 称	超 参 数	输入形状	输出形状
输入层	无	(128, 128, 3)	(128, 128, 3)
2D 零填充层	padding(1, 1)	(128, 128, 3)	(130, 130, 3)
2D 卷积层	filters=64, kernel_size=4, strides=2, padding='valid'	(130, 130, 3)	(64, 64, 64)
激活层	activation='leakyrelu', alpha=0.2	(64, 64, 64)	(64, 64, 64)
2D 零填充层	padding(1, 1)	(64, 64, 64)	(66, 66, 64)
2D 卷积层	filters=128, kernel_size=4, strides=2, padding='valid'	(66, 66, 64)	(32, 32, 128)
实例归一化层	axis=1	(32, 32, 128)	(32, 32, 128)
激活层	activation='leakyrelu', alpha=0.2	(32, 32, 128)	(32, 32, 128)
2D 零填充层	padding(1, 1)	(32, 32, 128)	(34, 34, 128)
2D 卷积层	filters=256, kernel_size=4, strides=2, padding='valid'	(34, 34, 128)	(16, 16, 256)
实例归一化层	axis=1	(16, 16, 256)	(16, 16, 256)
激活层	activation='leakyrelu', alpha=0.2	(16, 16, 256)	(16, 16, 256)
2D 零填充层	padding(1, 1)	(16, 16, 256)	(18, 18, 256)
2D 卷积层	filters=512, kernel_size=4, strides=2, padding='valid'	(18, 18, 256)	(8, 8, 512)
实例归一化层	axis=1	(8, 8, 512)	(8, 8, 512)
激活层	activation='leakyrelu', alpha=0.2	(8, 8, 512)	(8, 8, 512)
2D 零填充层	padding(1, 1)	(8, 8, 512)	(10, 10, 512)
2D 卷积层	filters=1, kernel_size=4, strides=1, padding='valid', activation='sigmoid'	(10, 10, 512)	(7, 7, 1)

判别网络返回形状为`(7, 7, 1)`的张量。介绍过了两个网络的具体架构，下面介绍训练 CycleGAN 所需的目标函数。



2D 零填充层在图像张量的上方、下方、左侧和右侧添加由 0 组成的行和列。

7.1.2　训练目标函数

和其他 GAN 类似，CycleGAN 有一个训练目标函数，需要在训练模型时最小化。该损失函数是下面两种损失的加权和。

(1) 对抗损失。
(2) 循环一致性损失。

下面详细介绍对抗损失和循环一致性损失。

1. 对抗损失

对抗损失是来自概率分布 A 或概率分布 B 的图像和生成网络生成的图像之间的损失。该网络涉及两个映射函数，都需要应用对抗损失。

映射 $G: X \to Y$ 对应的对抗损失形式如下。

$$L_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] \\ + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x))]$$

其中，x 是概率分布 A 领域的图像，y 是概率分布 B 领域的图像。判别网络 D_y 试图区分映射 G 生成的图像（即 $G(x)$）和概率分布 B 中的真实图像 y。判别网络 D_x 试图区分映射 F 生成的图像（即 $F(y)$）和概率分布 A 中的真实图像 x。G 旨在将对抗损失函数最小化，而对手 D 试图将该函数最大化。

2. 循环一致性损失

如果仅使用对抗损失，网络会将同样一组输入图像映射到目标领域的任一组随机组合的图像上。因此，获得的任何映射都可以习得一种类似于目标概率分布的输出，概率分布 x_i 和 y_i 之间就会有多种映射方式。循环一致性损失通过减少可能映射的数量来解决该问题。满足循环一致性的映射函数不仅可以将领域 A 中的图像 x 变换成领域 B 中的图像 y，还可以基于 y 生成原始图像 x。

正向循环一致性映射函数的形式如下。

$$x \to G(x) \to F(G(x)) \approx x$$

反向循环一致性映射函数的形式如下。

$$y \rightarrow F(y) \rightarrow G(F(y)) \approx y$$

循环一致性损失的公式如下。

$$L_{\text{cyc}}(G,F) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1]$$

如果使用循环一致性损失，那么通过 $F(G(x))$和 $G(F(y))$进行重构的图像会分别与 x 和 y 相似。

3. 完整目标函数

完整的目标函数是对抗损失和循环一致性损失的加权和，如下所示。

$$L(G,F,D_X,D_Y) = L_{\text{GAN}}(G,D_Y,X,Y) + L_{\text{GAN}}(F,D_X,Y,X) + \lambda L_{\text{cyc}}(G,F)$$

其中，$L_{\text{GAN}}(G, D_Y, X, Y)$是第一个对抗损失，$L_{\text{GAN}}(F, D_X, Y, X)$是第二个对抗损失。第一个对抗损失是基于对抗网络 A 和判别网络 B 计算的，第二个对抗损失是基于生成网络 B 和判别网络 A 计算的。

需要优化下面的函数，以训练 CycleGAN。

$$G^*, F^* = \arg\min_{G,F}\max_{D_X,D_Y} L(G,F,D_X,D_Y)$$

上式表明，如果要训练 CycleGAN，需要将生成网络损失最小化，而将判别网络损失最大化。优化后的网络能根据绘画生成照片。

7.2 创建项目

前面已经克隆或下载了本书所有章节的完整代码。其中目录 Chapter07 包含本章的完整代码。执行如下命令以创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter07，如下所示。

```
cd Chapter07
```

(3) 然后为本项目创建一个 Python 虚拟环境，如下所示。

```
virtualenv venv
virtualenv venv -p python3 # 创建一个使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建一个使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 启用新创建的虚拟环境，如下所示。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 执行如下命令，安装 requirements.txt 文件中列出的所需的库。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

这样就创建好了项目且安装了所需的依赖程序。下面处理数据集。

7.3　下载数据集

本章使用 monet2photo 数据集。该开源数据集由加州大学伯克利分校的 BAIR 提供，下载地址为：https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/monet2photo.zip。

下载之后解压到根目录下。

也可以执行以下命令，自动下载数据集。

```
wget
https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/monet2photo.zip
upzip monet2photo.zip
```

该命令会下载数据集，并解压到项目的根目录下。



monet2photo 数据集仅用于教学。如想商用，需获加州大学伯克利分校 BAIR 许可。我们未获数据集中图像的版权。

7.4　CycleGAN 的 Keras 实现

7.1 节讲过，CycleGAN 由一个生成网络和一个判别网络组成。下面编写这两个网络的实现代码。

开始编写代码之前，首先创建一个 Python 文件 main.py，并导入核心模块，如下所示。

```
from glob import glob
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from keras import Input, Model
from keras.layers import Conv2D, BatchNormalization, Activation, Add, \
    Conv2DTranspose, ZeroPadding2D, LeakyReLU
from keras.optimizers import Adam
from keras_contrib.layers import InstanceNormalization
from scipy.misc import imread, imresize
```

7.4.1 生成网络

前面介绍过生成网络的架构。下面使用 Keras 框架编写生成网络的各层，然后使用 Keras 框架的函数式 API 创建一个 Keras 模型。

用 Keras 实现生成网络的步骤如下。

(1) 首先定义生成网络所需的超参数，如下所示。

```
input_shape = (128, 128, 3)
residual_blocks = 6
```

(2) 然后创建一个输入层，为网络提供输入，如下所示。

```
input_layer = Input(shape=input_shape)
```

(3) 添加第 1 个卷积块，使用前面指定的超参数，如下所示。

```
x = Conv2D(filters=32, kernel_size=7, strides=1,
           padding="same")(input_layer)
x = InstanceNormalization(axis=1)(x)
x = Activation("relu")(x)
```

(4) 添加第 2 个卷积块，如下所示。

```
x = Conv2D(filters=64, kernel_size=3, strides=2, padding="same")(x)
x = InstanceNormalization(axis=1)(x)
x = Activation("relu")(x)
```

(5) 添加第 3 个卷积块，如下所示。

```
x = Conv2D(filters=128, kernel_size=3, strides=2,
           padding="same")(x)
x = InstanceNormalization(axis=1)(x)
x = Activation("relu")(x)
```

(6) 定义一个残差块，如下所示。

```
def residual_block(x):
    """
    残差块
```

```
    """
    res = Conv2D(filters=128, kernel_size=3, strides=1,
                 padding="same")(x)
    res = BatchNormalization(axis=3, momentum=0.9,
                             epsilon=1e-5)(res)
    res = Activation('relu')(res)

    res = Conv2D(filters=128, kernel_size=3, strides=1,
                 padding="same")(res)
    res = BatchNormalization(axis=3, momentum=0.9,
    epsilon=1e-5)(res)

        return Add()([res, x])
```

然后使用 `residual_block()` 函数向模型添加 6 个残差块，如下所示。

```
for _ in range(residual_blocks):
    x = residual_block(x)
```

(7) 接着添加一个上采样块，如下所示。

```
x = Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                    padding='same', use_bias=False)(x)
x = InstanceNormalization(axis=1)(x)
x = Activation("relu")(x)
```

(8) 再添加一个上采样块，如下所示。

```
x = Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                    padding='same', use_bias=False)(x)
x = InstanceNormalization(axis=1)(x)
x = Activation("relu")(x)
```

(9) 最后，添加输出卷积层，如下所示。

```
x = Conv2D(filters=3, kernel_size=7, strides=1, padding="same")(x)
           output = Activation('tanh')(x)
```

这是生成网络的最后一层，生成形状为 `(128, 128, 3)` 的图像。

(10) 指定网络的输出和输入以创建 Keras 模型。

```
model = Model(inputs=[input_layer], outputs=[output])
```

生成网络的完整代码如下。

```
def build_generator():
    """
    使用下面定义的超参数值创建一个生成网络
    """
    input_shape = (128, 128, 3)
    residual_blocks = 6
    input_layer = Input(shape=input_shape)
```

```
    # 第 1 个卷积块
    x = Conv2D(filters=32, kernel_size=7, strides=1, padding="same")(input_layer)
    x = InstanceNormalization(axis=1)(x)
    x = Activation("relu")(x)

    # 第 2 个卷积块
    x = Conv2D(filters=64, kernel_size=3, strides=2, padding="same")(x)
    x = InstanceNormalization(axis=1)(x)
    x = Activation("relu")(x)

    # 第 3 个卷积块
    x = Conv2D(filters=128, kernel_size=3, strides=2, padding="same")(x)
    x = InstanceNormalization(axis=1)(x)
    x = Activation("relu")(x)

    # 残差块
    for _ in range(residual_blocks):
        x = residual_block(x)

    # 上采样块
    # 第 1 个上采样块
    x = Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                        padding='same', use_bias=False)(x)
    x = InstanceNormalization(axis=1)(x)
    x = Activation("relu")(x)

    # 第 2 个上采样块
    x = Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                        padding='same', use_bias=False)(x)
    x = InstanceNormalization(axis=1)(x)
    x = Activation("relu")(x)

    # 最终卷积层
    x = Conv2D(filters=3, kernel_size=7, strides=1, padding="same")(x)
              output = Activation('tanh')(x)

    model = Model(inputs=[input_layer], outputs=[output])
    return model
```

这样就创建好了生成网络的 Keras 模型。下面创建判别网络的 Keras 模型。

7.4.2 判别网络

前面介绍过判别网络的架构。下面使用 Keras 框架编写判别网络各层，然后使用 Keras 框架的函数式 API 创建一个 Keras 模型。

用 Keras 实现判别网络的步骤如下。

(1) 首先定义判别网络所需的超参数，如下所示。

```
input_shape = (128, 128, 3)
hidden_layers = 3
```

(2) 然后添加一个输入层，为网络提供输入，如下所示。

```
input_layer = Input(shape=input_shape)
```

(3)接着添加一个 2D 零填充层，如下所示。

```
x = ZeroPadding2D(padding=(1, 1))(input_layer)
```

该层对输入张量在 x 轴和 y 轴上同时进行填充。

(4) 然后使用前面指定的超参数添加一个卷积块，如下所示。

```
x = Conv2D(filters=64, kernel_size=4, strides=2, padding="valid")(x)
x = LeakyReLU(alpha=0.2)(x)
```

(5) 再添加一个 2D 零填充层，如下所示。

```
x = ZeroPadding2D(padding=(1, 1))(x)
```

(6) 然后使用前面指定的超参数添加 3 个卷积块，如下所示。

```
for i in range(1, hidden_layers + 1):
    x = Conv2D(filters=2 ** i * 64, kernel_size=4, strides=2,
               padding="valid")(x)
    x = InstanceNormalization(axis=1)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = ZeroPadding2D(padding=(1, 1))(x)
```

每个卷积块包含两个卷积层、一个实例归一化层、一个激活层和一个 2D 零填充层。

(7) 为网络添加最终的（输出）卷积层，如下所示。

```
output = Conv2D(filters=1, kernel_size=4, strides=1,
                activation="sigmoid")(x)
```

(8) 最后，指定网络的输入和输出以创建 Keras 模型。

```
model = Model(inputs=[input_layer], outputs=[output])
```

判别网络的完整代码如下。

```
def build_discriminator():
    """
    使用下面定义的超参数值创建一个判别网络
    """
    input_shape = (128, 128, 3)
    hidden_layers = 3

    input_layer = Input(shape=input_shape)
```

```
    x = ZeroPadding2D(padding=(1, 1))(input_layer)

    # 第 1 个卷积块
    x = Conv2D(filters=64, kernel_size=4, strides=2, padding="valid")(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = ZeroPadding2D(padding=(1, 1))(x)

    # 3 个隐藏卷积块
    for i in range(1, hidden_layers + 1):
        x = Conv2D(filters=2 ** i * 64, kernel_size=4, strides=2,
                   padding="valid")(x)
        x = InstanceNormalization(axis=1)(x)
        x = LeakyReLU(alpha=0.2)(x)

        x = ZeroPadding2D(padding=(1, 1))(x)

    # 最终的卷积层
    output = Conv2D(filters=1, kernel_size=4, strides=1, activation="sigmoid")(x)

    model = Model(inputs=[input_layer], outputs=[output])
    return model
```

这样就创建好了判别网络的 Keras 模型。下面训练网络。

7.5 训练 CycleGAN

7.1 节介绍了训练目标函数，并且两个网络的 Keras 模型也创建好了。训练 CycleGAN 包含多个步骤，如下所示。

(1) 加载数据集。
(2) 创建生成网络和判别网络。
(3) 按特定轮数训练网络。
(4) 绘制损失图。
(5) 生成新图像。

在开始训练之前，首先定义一些核心变量，如下所示。

```
data_dir = "/Path/to/dataset/directory/*.*"
batch_size = 1
epochs = 500
```

7.5.1 加载数据集

加载数据集的步骤如下。

(1) 首先使用 glob 模块，创建一个由图像路径组成的列表，如下所示。

```
imagesA = glob(data_dir + '/testA/*.*')
imagesB = glob(data_dir + '/testB/*.*')
```

由于数据来自领域 A 和领域 B，因此创建了两个列表。

(2) 然后迭代列表。在循环内部加载图像并缩放，再在水平方向上翻转，如下所示。

```
allImagesA = []
allImagesB = []

# 迭代列表
for index, filename in enumerate(imagesA):
# 加载图像
imgA = imread(filename, mode='RGB')
imgB = imread(imagesB[index], mode='RGB')
# 缩放图像
imgA = imresize(imgA, (128, 128))
imgB = imresize(imgB, (128, 128))
# 在水平方向上随机翻转图像
if np.random.random() > 0.5:
    imgA = np.fliplr(imgA)
    imgB = np.fliplr(imgB)

allImagesA.append(imgA)
allImagesB.append(imgB)
```

(3) 接着将图像归一化，将像素值转换到介于–1 到 1 之间，如下所示。

```
# 将图像归一化
allImagesA = np.array(allImagesA) / 127.5 - 1.
allImagesB = np.array(allImagesB) / 127.5 - 1.
```

加载数据集的完整代码如下。

```
def load_images(data_dir):
    imagesA = glob(data_dir + '/testA/*.*')
    imagesB = glob(data_dir + '/testB/*.*')

    allImagesA = []
    allImagesB = []

    for index, filename in enumerate(imagesA):
        # 加载图像
        imgA = imread(filename, mode='RGB')
        imgB = imread(imagesB[index], mode='RGB')
        # 缩放图像
        imgA = imresize(imgA, (128, 128))
        imgB = imresize(imgB, (128, 128))
        # 在水平方向上随机翻转图像
        if np.random.random() > 0.5:
            imgA = np.fliplr(imgA)
            imgB = np.fliplr(imgB)

        allImagesA.append(imgA)
        allImagesB.append(imgB)
```

```
    # 将图像归一化
    allImagesA = np.array(allImagesA) / 127.5 - 1.

    allImagesB = np.array(allImagesB) / 127.5 - 1.

    return allImagesA, allImagesB
```

上面的函数返回两个 NumPy ndarray。在开始训练之前，首先使用该函数加载图像并进行预处理。

7.5.2 构建并编译网络

下面构建核心网络，并为训练做准备。操作步骤如下。

(1) 首先定义训练所需的优化器，代码如下。

```
# 定义共用的优化器
common_optimizer = Adam(0.0002, 0.5)
```

选择 Adam 优化器，设置学习速率为 0.0002、`beta_1` 值为 0.5。

(2) 创建判别网络，代码如下。

```
discriminatorA = build_discriminator()
discriminatorB = build_discriminator()
```

前面讲过，CycleGAN 拥有两个判别网络。

(3) 然后编译两个判别网络，如下所示。

```
discriminatorA.compile(loss='mse', optimizer=common_optimizer,
                       metrics=['accuracy'])
discriminatorB.compile(loss='mse', optimizer=common_optimizer,
                       metrics=['accuracy'])
```

使用 `mse` 作为损失函数、`accuracy` 作为度量，编译网络。

(4) 接着创建生成网络 A（generatorAToB）和生成网络 B（generatorBToA）。生成网络 A 的输入是数据集 A 中的真实图像（realA），输出为重构的图像（fakeB）。生成网络 B 的输入是数据集 B 中的真实图像（realB），输出是重构的图像（fakeA），如下所示。

```
generatorAToB = build_generator()
generatorBToA = build_generator()
```

7.1.1 节讲过，CycleGAN 拥有两个生成网络。generatorAToB 将图像从领域 A 转换到领域 B，而 generatorBToA 将图像从领域 B 转换到领域 A。

这样就创建好了两个生成网络和两个判别网络。下面创建并编译对抗网络。

创建并编译对抗网络

对抗网络是一个组合网络，在一个 Keras 模型中使用了全部 4 个网络。创建对抗网络旨在训练生成网络。训练对抗网络时，锁定判别网络，只训练生成网络。下面创建一个具有理想功能的对抗网络。

(1) 首先为网络创建两个输入层，如下所示。

```
inputA = Input(shape=(128, 128, 3))
inputB = Input(shape=(128, 128, 3))
```

两个输入都接收维度为`(128, 128, 3)`的图像。这些输入是符号输入变量，并未实际取值，用于创建 Keras 模型（TensorFlow 图）。

(2) 然后使用生成网络生成假图像，如下所示。

```
generatedB = generatorAToB(inputA)
generatedA = generatorBToA(inputB)
```

使用符号输入层生成图像。

(3) 接着使用另一组生成网络重构原始图像，如下所示。

```
reconstructedA = generatorBToA(generatedB)
reconstructedB = generatorAToB(generatedA)
```

(4) 使用生成网络生成假图像，如下所示。

```
generatedAId = generatorBToA(inputA)
generatedBId = generatorAToB(inputB)
```

生成网络 A（generatorAToB）将图像从领域 A 转换到领域 B，而生成网络 B（generatorBToA）将图像从领域 B 转换到领域 A。

(5) 然后将两个判别网络设置为“不可训练”，如下所示。

```
discriminatorA.trainable = False
discriminatorB.trainable = False
```

在对抗网络中，不训练判别网络。

(6) 使用判别网络估测每个生成图像是真是假，如下所示。

```
probsA = discriminatorA(generatedA)
probsB = discriminatorB(generatedB)
```

(7) 创建一个 Keras 模型，并指定网络的输入和输出，如下所示。

```
adversarial_model = Model(inputs=[inputA, inputB],
                          outputs=[probsA, probsB, reconstructedA,
                          reconstructedB, generatedAId, generatedBId])
```

对抗网络接收两个输入值（都是张量），返回 6 个输出值（都是张量）。

(8) 然后编译对抗网络，如下所示。

```
adversarial_model.compile(loss=['mse', 'mse', 'mae', 'mae', 'mae', 'mae'],
                          loss_weights=[1, 1, 10.0, 10.0, 1.0, 1.0],
                          optimizer=common_optimizer)
```

对抗网络返回 6 个值，需要为每个输出值指定损失函数。对于前两个值，使用**均方误差**损失，因为这是**对抗**损失的一部分；对于后面 4 个值，使用**平均绝对误差**损失，因为这是循环一致性损失的一部分。6 个损失的权重值分别为 1、1、10.0、10.0、1.0 和 1.0。使用 `common_optimizer` 训练网络。

这样就创建好了对抗网络的 Keras 模型。关于 Keras 模型的工作原理，请参考 TensorFlow 图及其功能的相关文档。

开始训练之前，按照如下两个关键步骤进行操作。稍后会用到 TensorBoard。

添加 TensorBoard 以存储用于进行可视化的损失和图，如下所示。

```
tensorboard = TensorBoard(log_dir="logs/{}".format(time.time()),
                          write_images=True, write_grads=True,
                          write_graph=True)
tensorboard.set_model(generatorAToB)
tensorboard.set_model(generatorBToA)
tensorboard.set_model(discriminatorA)
tensorboard.set_model(discriminatorB)
```

创建一个四维数组，所有值都为 1，代表真实标签。类似地，再创建一个四维数组，所有值都为 0，代表虚假标签，如下所示。

```
real_labels = np.ones((batch_size, 7, 7, 1))
fake_labels = np.zeros((batch_size, 7, 7, 1))
```

使用 NumPy 的 `ones()` 函数和 `zeros()` 函数创建所需的数组。至此，核心部分就准备好了，下面开始进行训练。

7.5.3 开始训练

根据下面各步骤，按指定轮数训练网络。

(1) 首先加载两个领域的数据集，如下所示。

```
imagesA, imagesB = load_images(data_dir=data_dir)
```

`load_images` 函数是在 7.5.1 节定义的。

(2) 然后创建一个 `for` 循环，其运行次数和指定轮数相同，如下所示。

```
for epoch in range(epochs):
    print("Epoch:{}".format(epoch))
```

(3) 创建两个列表，用于存储所有小批次的损失，如下所示。

```
dis_losses = []
gen_losses = []
```

(4) 计算每轮训练循环中的小批次数量，如下所示。

```
num_batches = int(min(imagesA.shape[0], imagesB.shape[0]) / batch_size)
print("Number of batches:{}".format(num_batches))
```

(5) 接着在训练轮循环内部再创建一个循环，其运行次数与 `num_batches` 指定的数量相同，如下所示。

```
for index in range(num_batches):
    print("Batch:{}".format(index))
```

训练判别网络和对抗网络的完整代码都在该循环中。

1. 训练判别网络

以下代码接着前面的代码。训练判别网络的流程如下。

(1) 首先从两个领域分别采样一小批次图像，如以下代码所示。

```
batchA = imagesA[index * batch_size:(index + 1) * batch_size]
batchB = imagesB[index * batch_size:(index + 1) * batch_size]
```

(2) 然后使用生成网络生成假图像，如下所示。

```
generatedB = generatorAToB.predict(batchA)
generatedA = generatorBToA.predict(batchB)
```

(3) 接着使用真实图像和（生成网络生成的）虚假图像训练判别网络 A，如下所示。

```
dALoss1 = discriminatorA.train_on_batch(batchA, real_labels)
dALoss2 = discriminatorA.train_on_batch(generatedA, fake_labels)
```

该步骤使用一小批次真实图像和虚假图像训练判别网络 A，使之小幅优化。

(4) 然后使用真实图像和虚假图像训练判别网络 B，如下所示。

```
dBLoss1 = discriminatorB.train_on_batch(batchB, real_labels)
dbLoss2 = discriminatorB.train_on_batch(generatedB, fake_labels)
```

(5) 计算判别网络的总损失值，如下所示。

```
d_loss = 0.5 * np.add(0.5 * np.add(dALoss1, dALoss2),
                      0.5 * np.add(dBLoss1, dbLoss2))
```

添加好了训练判别网络的代码，下面通过训练对抗网络来训练生成网络。

2. 训练对抗网络

训练对抗网络需要使用输入值和真实值。网络的输入值是 batchA 和 batchB。真实值是 real_labels、real_labels、batchA、batchB、batchA 和 batchB，如下所示。

```
g_loss = adversarial_model.train_on_batch([batchA, batchB],
                                          [real_labels, real_labels,
                                           batchA, batchB, batchA, batchB])
```

这一步训练生成网络，不训练判别网络。

在每个小批次的迭代（循环）完成之后，将损失存储在名为 dis_losses 和 gen_losses 的列表中，如下所示。

```
dis_losses.append(d_loss)
gen_losses.append(g_loss)
```

每训练 10 轮，使用生成网络生成一组图像。

```
# 每训练 10 轮，采样图像并保存
if epoch % 10 == 0:
    # 获取一批次测试数据
    batchA, batchB = load_test_batch(data_dir=data_dir, batch_size=2)
    # 生成图像
    generatedB = generatorAToB.predict(batchA)
    generatedA = generatorBToA.predict(batchB)
    # 获取重构图像
    reconsA = generatorBToA.predict(generatedB)
    reconsB = generatorAToB.predict(generatedA)
    # 保存原始图像、生成图像和重构图像
    for i in range(len(generatedA)):
        save_images(originalA=batchA[i], generatedB=generatedB[i],
                    recosntructedA=reconsA[i],
                    originalB=batchB[i], generatedA=generatedA[i],
                    reconstructedB=reconsB[i],
                    path="results/gen_{}_{}".format(epoch, i))
```

将上面的代码块放入轮循环中。每训练 10 轮，生成一批次假图像并保存到结果目录中。

然后将平均损失存储到 TensorBoard 中，用于可视化，包括生成网络的平均损失和判别网络的平均损失。如下所示。

```
write_log(tensorboard, 'discriminator_loss', np.mean(dis_losses), epoch)
write_log(tensorboard, 'generator_loss', np.mean(gen_losses), epoch)
```

将上面的代码块放入轮循环中。

7.5.4 保存模型

在 Keras 中，保存模型只需一行代码，如下所示。

```
# 指定生成网络 A 模型的路径
generatorAToB.save("directory/for/the/generatorAToB/model.h5")

# 指定生成网络 B 模型的路径
generatorBToA.save("directory/for/the/generatorBToA/model.h5")
```

类似地，保存判别网络模型的代码如下。

```
# 指定判别网络 A 模型的路径
discriminatorA.save("directory/for/the/discriminatorA/model.h5")

# 指定判别网络 B 模型的路径
discriminatorB.save("directory/for/the/discriminatorB/model.h5")
```

7.5.5　生成图像可视化

100 轮训练后，生成网络会开始生成比较不错的图像。下面看一下这些生成的图像。

10 轮训练后，图像如图 7-2 所示。

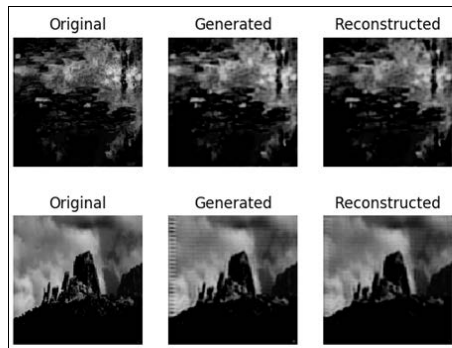


图 7-2　10 轮训练后生成的图像

20 轮训练后，图像如图 7-3 所示。

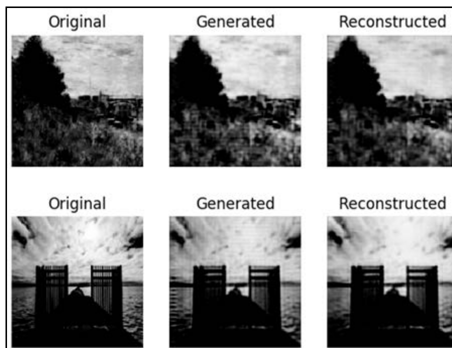


图 7-3　20 轮训练后生成图像

建议将网络训练 1000 轮。一切顺利的话，1000 轮训练后，生成网络会开始生成逼真的图像。

7.5.6　损失可视化

启动 TensorBoard 服务器，将训练损失可视化。如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分包含了两种损失的曲线图，如下所示。

判别网络的损失图如图 7-4 所示。

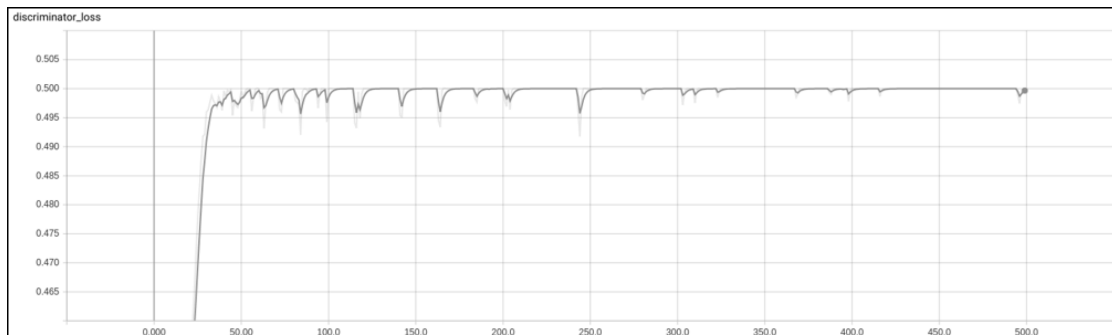


图 7-4　判别网络的损失图

生成网络的损失图如图 7-5 所示。

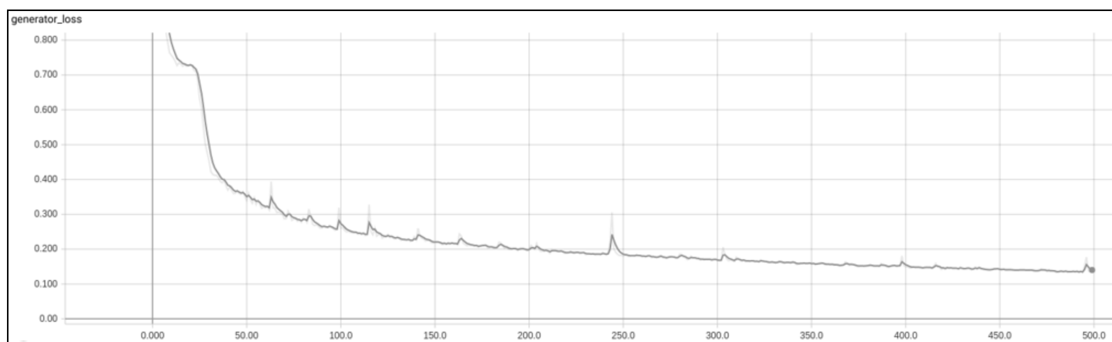


图 7-5　生成网络的损失图

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

7.5.7　图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 7-6）。

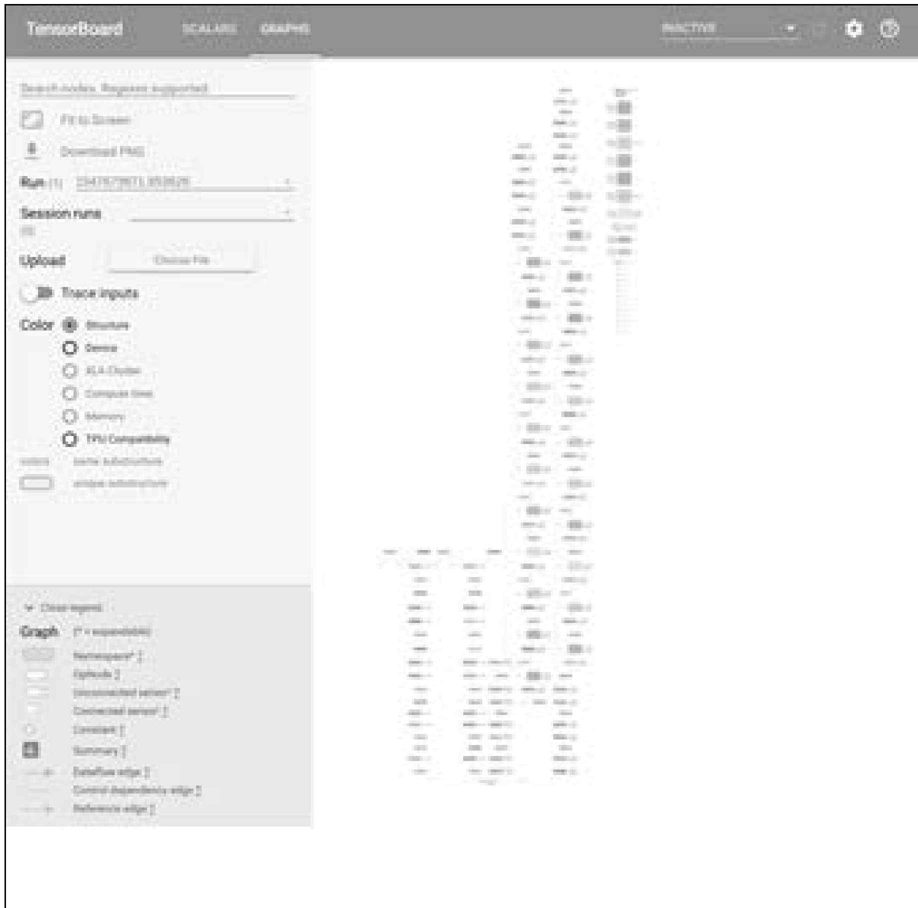


图 7-6　各图中的张量和不同运算的流

7.6　CycleGAN 的实际应用

CycleGAN 有很多实际应用。本章使用 CycleGAN 将绘画转换为照片。CycleGAN 还可应用于以下场景。

- ❑ **风格变换**：比如将照片转换为绘画（或者反过来）、将马的照片转换为斑马的照片（或者反过来）以及将橙子的照片转换为苹果的照片（或者反过来）。

- **照片增强**：CycleGAN 可用于提高照片质量。
- **季节转换**：比如将冬天拍摄的照片转换为夏天拍摄的照片（或者反过来）。
- **游戏风格迁移**：CycleGAN 可用于将 A 游戏的风格迁移到 B 游戏。

7.7 小结

本章介绍了如何使用 CycleGAN 将绘画转换为照片。首先简单介绍了 CycleGAN，探讨了 CycleGAN 涉及的网络架构，然后介绍了训练 CycleGAN 所需的不同损失函数，接着使用 Keras 框架实现了 CycleGAN，随后在 monet2photo 数据集上训练 CycleGAN，并且对生成的图像、损失，以及不同网络的图都进行了可视化，最后介绍了 CycleGAN 的实际应用。

下一章会介绍用于图像对图像变换的 pix2pix 网络，以及用于图像变换的 cGAN。

7.8 延伸阅读

CycleGAN 用途广泛。可以参考以下文章，了解 CycleGAN 的更多用途。

- “Turning Fortnite into PUBG with Deep Learning (CycleGAN)”
- “GAN — CycleGAN (Playing magic with pictures)”
- “Introduction to CycleGANs”
- “Understanding and Implementing CycleGAN in TensorFlow”

第 8 章

使用 cGAN 实现图像对图像变换

8

pix2pix 是一种 GAN，用于进行图像对图像变换，即将图像从一种表示变换为另一种表示。pix2pix 学习从输入图像到输出图像的一种映射，可用于将黑白图像转换为彩色图像、将草图转换为照片、将白天的图像转换为夜间的图像、将卫星图像转化为地图图像等。pix2px 网络最初是由 Phillip Isola、Jun-Yan Zhu、Tinghui Zhou 和 Alexei A. Efros 在论文“Image-to-Image Translation with Conditional Adversarial Networks”中提出的。

本章将讨论以下主题。

- pix2pix 网络简介
- pix2pix 网络架构
- 数据收集和准备
- pix2pix 的 Keras 实现
- 目标函数
- 训练 pix2pix
- 评估训练好的模型
- pix2pix 网络的实际应用

8.1 pix2pix 简介

pix2pix 是 cGAN 的一个变体。第 3 章介绍过 cGAN，后续内容将基于它展开。pix2pix 可以运用无监督机器学习技术进行图像对图像变换。训练完成后，pix2pix 可以将图像从领域 A 变换到领域 B。普通 CNN 也可以进行图像对图像变换，但是不能生成清晰逼真的图像，而 pix2pix 可以。本章会使用 pix2pix 将建筑立面标记图转换为建筑立面图像。首先介绍 pix2pix 的架构。

8.1.1 pix2pix 架构

和其他 GAN 类似，pix2pix 由一个生成网络和一个判别网络组成。生成网络的架构受到了 U-Net 架构的启发，而判别网络的架构受到了 PatchGAN 架构的启发。这两个网络都是深度卷积神经网络。下面深入探讨 pix2pix。

1. 生成网络

前面提过，生成网络深受 U-Net 架构的启发。U-Net 架构和自编码网络的架构非常相似，一个主要区别是 U-Net 网络在编码网络和解码网络的各层之间有跳跃连接，而自编码网络没有。U-Net 网络由编码网络和解码网络组成。图 8-1 展示了 U-Net 的基本架构。

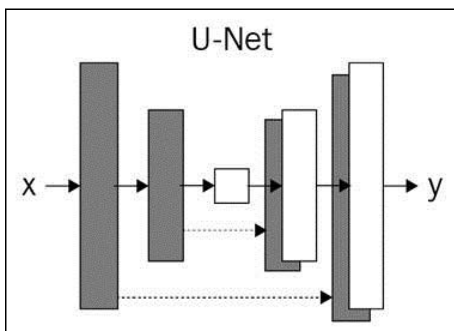


图 8-1　U-Net 的基本架构

上图展示了 U-Net 基本架构。可以看到，第一层的输出直接和最后一层合并，第二层的输出和倒数第二层合并，以此类推。如果总层数为 n，那么在编码网络的第 i 层和解码网络的第$(n-i)$层之间就会有跳跃连接，对于所有层皆是如此。下面详细介绍这两个网络。

- **编码网络**

编码网络是生成网络的前半部分，由 8 个**卷积块**组成，其配置见表 8-1。

表　8-1

层　名　称	超　参　数	输入形状	输出形状
第 1 个 2D 卷积层	filters=64, kernel_size=4, strides=2, padding='same',	(256, 256, 1)	(128, 128, 64)
激活层	activation='leakyrelu', alpha=0.2	(128, 128, 64)	(128, 128, 64)
第 2 个 2D 卷积层	filters=128, kernel_size=4, strides=2, padding='same',	(128, 128, 64)	(64, 64, 128)
批归一化层	无	(64, 64, 128)	(64, 64, 128)
激活层	activation='leakyrelu', alpha=0.2	(64, 64, 128)	(64, 64, 128)

（续）

层　名　称	超　参　数	输入形状	输出形状
第 3 个 2D 卷积层	filters=256, kernel_size=4, strides=2, padding='same',	(64, 64, 128)	(32, 32, 256)
批归一化层	无	(32, 32, 256)	(32, 32, 256)
激活层	activation='leakyrelu', alpha=0.2	(32, 32, 256)	(32, 32, 256)
第 4 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(32, 32, 256)	(16, 16, 512)
批归一化层	无	(16, 16, 512)	(16, 16, 512)
激活层	activation='leakyrelu', alpha=0.2	(16, 16, 512)	(16, 16, 512)
第 5 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(16, 16, 512)	(8, 8, 512)
批归一化层	无	(8, 8, 512)	(8, 8, 512)
激活层	activation='leakyrelu', alpha=0.2	(8, 8, 512)	(8, 8, 512)
第 6 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(8, 8, 512)	(4, 4, 512)
批归一化层	无	(4, 4, 512)	(4, 4, 512)
激活层	activation='leakyrelu', alpha=0.2	(4, 4, 512)	(4, 4, 512)
第 7 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(4, 4, 512)	(2, 2, 512)
批归一化层	无	(2, 2, 512)	(2, 2, 512)
激活层	activation='leakyrelu', alpha=0.2	(2, 2, 512)	(2, 2, 512)
第 8 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(2, 2, 512)	(1, 1, 512)
批归一化层	无	(1, 1, 512)	(1, 1, 512)
激活层	activation='leakyrelu', alpha=0.2	(1, 1, 512)	(1, 1, 512)

编码网络之后是解码网络。下面介绍解码网络的架构。

- **解码网络**

生成网络中的解码网络由 8 个**上采样卷积块**组成，其配置见表 8-2。

表　8-2

层　名　称	超　参　数	输入形状	输出形状
第 1 个 2D 上采样卷积层	size=(2, 2)	(1, 1, 512)	(2, 2, 512)
2D 卷积层	filters=512, kernel_size=4, strides=1, padding='same',	(2, 2, 512)	(2, 2, 512)

（续）

层 名 称	超 参 数	输入形状	输出形状
批归一化层	无	(2, 2, 512)	(2, 2, 512)
随机失活层	dropout=0.5	(2, 2, 512)	(2, 2, 512)
拼接层（编码网络的第 7 层）	axis=3	(2, 2, 512)	(2, 2, 1024)
激活层	activation='relu'	(2, 2, 1024)	(2, 2, 1024)
第 2 个 2D 上采样卷积层	size=(2, 2)	(2, 2, 1024)	(4, 4, 1024)
2D 卷积层	filters=1024, kernel_size=4, strides=1, padding='same',	(4, 4, 1024)	(4, 4, 1024)
批归一化层	无	(4, 4, 1024)	(4, 4, 1024)
随机失活层	dropout=0.5	(4, 4, 1024)	(4, 4, 1024)
拼接层（编码网络的第 6 层）	axis=3	(4, 4, 1024)	(4, 4, 1536)
激活层	activation='relu'	(4, 4, 1536)	(4, 4, 1536)
第 3 个 2D 上采样卷积层	size=(2, 2)	(4, 4, 1536)	(8, 8, 1536)
2D 卷积层	filters=1024, kernel_size=4, strides=1, padding='same',	(8, 8, 1536)	(8, 8, 1024)
批归一化层	无	(8, 8, 1024)	(8, 8, 1024)
随机失活层	dropout=0.5	(8, 8, 1024)	(8, 8, 1024)
拼接层（编码网络的第 5 层）	axis=3	(8, 8, 1024)	(8, 8, 1024)
激活层	activation='relu'	(8, 8, 1536)	(8, 8, 1536)
第 4 个 2D 上采样卷积层	size=(2, 2)	(8, 8, 1536)	(16, 16, 1536)
2D 卷积层	filters=1024, kernel_size=4, strides=1, padding='same',	(16, 16, 1536)	(16, 16, 1024)
批归一化层	无	(16, 16, 1024)	(16, 16, 1024)
拼接层（编码网络的第 4 层）	axis=3	(16, 16, 1024)	(16, 16, 1536)
激活层	activation='relu'	(16, 16, 1536)	(16, 16, 1536)
第 5 个 2D 上采样卷积层	size=(2, 2)	(16, 16, 1536)	(32, 32, 1536)
2D 卷积层	filters=1024, kernel_size=4, strides=1, padding='same',	(32, 32, 1536)	(32, 32, 1024)
批归一化层	无	(32, 32, 1024)	(32, 32, 1024)
拼接层（编码网络的第 3 层）	axis=3	(32, 32, 1024)	(32, 32, 1280)
激活层	activation='relu'	(32, 32, 1280)	(32, 32, 1280)
第 6 个 2D 上采样卷积层	size=(2, 2)	(64, 64, 1280)	(64, 64, 1280)

（续）

层　名　称	超　参　数	输入形状	输出形状
2D 卷积层	filters=512, kernel_size=4, strides=1, padding='same',	(64, 64, 1280)	(64, 64, 512)
批归一化层	无	(64, 64, 512)	(64, 64, 512)
拼接层（编码网络的第 2 层）	axis=3	(64, 64, 512)	(64, 64, 640)
激活层	activation='relu'	(64, 64, 640)	(64, 64, 640)
第 7 个 2D 上采样卷积层	size=(2, 2)	(64, 64, 640)	(128, 128, 640)
2D 卷积层	filters=256, kernel_size=4, strides=1, padding='same',	(128, 128, 640	(128, 128, 256)
批归一化层	无	(128, 128, 256)	(128, 128, 256)
拼接层（编码网络的第 1 层）	axis=3	(128, 128, 256)	(128, 128, 320)
激活层	activation='relu'	(128, 128, 320)	(128, 128, 320)
第 8 个 2D 上采样卷积层	size=(2, 2)	(128, 128, 320	(256, 256, 320)
2D 卷积层	filters=1, kernel_size=4, strides=1, padding='same',	(256, 256, 320	(256, 256, 1)
激活层	activation='tanh'	(256, 256, 1)	(256, 256, 1)

生成网络中有 7 个跳跃连接，定义如下。

- 编码网络中第 1 块的输出和解码网络的第 7 块进行拼接。
- 编码网络中第 2 块的输出和解码网络的第 6 块进行拼接。
- 编码网络中第 3 块的输出和解码网络的第 5 块进行拼接。
- 编码网络中第 4 块的输出和解码网络的第 4 块进行拼接。
- 编码网络中第 5 块的输出和解码网络的第 3 块进行拼接。
- 编码网络中第 6 块的输出和解码网络的第 2 块进行拼接。
- 编码网络中第 7 块的输出和解码网络的第 1 块进行拼接。

所有拼接都是沿通道轴进行的。编码网络的最后一层将张量传递给解码网络的第一层。编码网络的最后一块和解码网络的最后一块之间并无拼接。

生成网络由上述两个网络组成。简单说来，编码网络是一个下采样网络，解码网络是一个上采样网络。编码网络将维度是`(256, 256, 1)`的图像下采样为维度是`(1, 1, 512)`的内部表示，而解码网络将维度是`(1, 1, 512)`的内部表示上采样为维度是`(256, 256, 1)`的输出图像。



8.4 节会详细介绍生成网络的架构。

2. 判别网络架构

pix2pix 的判别网络架构受到了 PatchGAN 架构的启发。PatchGAN 包含 8 个卷积块，如表 8-3 所示。

表 8-3

层 名 称	超 参 数	输入形状	输出形状
第 1 个 2D 卷积层	filters=64, kernel_size=4, strides=2, padding='same',	(256, 256, 1)	(256, 256, 64)
激活层	activation='leakyrelu', alpha=0.2	(128, 128, 64)	(128, 128, 64)
第 2 个 2D 卷积层	filters=128, kernel_size=4, strides=2, padding='same',	(128, 128, 64)	(64, 64, 128)
批归一化层	无	(64, 64, 128)	(64, 64, 128)
激活层	activation='leakyrelu', alpha=0.2	(64, 64, 128)	(64, 64, 128)
第 3 个 2D 卷积层	filters=256, kernel_size=4, strides=2, padding='same',	(64, 64, 128)	(32, 32, 256)
批归一化层	无	(32, 32, 256)	(32, 32, 256)
激活层	activation='leakyrelu', alpha=0.2	(32, 32, 256)	(32, 32, 256)
第 4 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(32, 32, 256)	(16, 16, 512)
批归一化层	无	(16, 16, 512)	(16, 16, 512)
激活层	activation='leakyrelu', alpha=0.2	(16, 16, 512)	(16, 16, 512)
第 5 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(16, 16, 512)	(8, 8, 512)
批归一化层	无	(8, 8, 512)	(8, 8, 512)
激活层	activation='leakyrelu', alpha=0.2	(8, 8, 512)	(8, 8, 512)
第 6 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(8, 8, 512)	(4, 4, 512)
批归一化层	无	(4, 4, 512)	(4, 4, 512)
激活层	activation='leakyrelu', alpha=0.2	(4, 4, 512)	(4, 4, 512)
第 7 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(4, 4, 512)	(2, 2, 512)
批归一化层	无	(2, 2, 512)	(2, 2, 512)
激活层	activation='leakyrelu', alpha=0.2	(2, 2, 512)	(2, 2, 512)
第 8 个 2D 卷积层	filters=512, kernel_size=4, strides=2, padding='same',	(4, 4, 512)	(1, 1, 512)
批归一化层	无	(1, 1, 512)	(1, 1, 512)
激活层	activation='leakyrelu', alpha=0.2	(1, 1, 512)	(1, 1, 512)
扁平化层	无	(1, 1, 512)	(512,)
全连接层	units=2, activation='softmax'	(1, 1, 512)	(2,)

上表展现了判别网络的架构和配置。其中，扁平化层将张量转换成一维数组。



8.4 节将会介绍判别网络的余下各层。

介绍过了两个网络的架构和配置，下面介绍训练 pix2pix 所需的目标函数。

8.1.2　训练目标函数

pix2pix 是一种 cGAN，其目标函数形式如下。

$$L_{\mathrm{cGAN}}(G,D)=E_{x,y}[\log D(x,y)]+E_{x,z}[\log(1-D(x,G(x,z)))]$$

网络 G（生成网络）试图将上面的函数最小化，而对手 D（判别网络）试图将其最大化。

普通 GAN 的目标函数如下，便于和 cGAN 的目标函数对比。

$$L_{\mathrm{GAN}}(G,D)=E_{y}[\log D(y)]+E_{x,z}[\log(1-D(G(x,z)))]$$

可在目标函数中添加一个 L1 损失函数，以避免图像过于模糊。L1 损失函数形式如下。

$$L_{L1}(G)=E_{x,y,z}[\| y-G(x,z)\|_1]$$

其中，y 是原始图像，$G(x,z)$是生成网络生成的图像。L1 损失的计算过程是，取原始图像和生成图像的每个像素值之间的绝对差，然后对所有绝对差求和。

pix2pix 的最终目标函数如下。

$$G^*=\arg\min_G\max_D L_{\mathrm{cGAN}}(G,D)+\lambda L_{L1}(G)$$

这是 cGAN 的损失函数和 L1 损失函数的加权和。

以上是 pix2pix 网络的基础知识。在使用 Keras 实现 pix2pix 之前，首先创建项目。

8.2　创建项目

前面已经克隆或下载了本书所有章节的完整代码。其中目录 Chapter08 包含本章的完整代码。执行如下命令以创建项目。

(1) 首先访问父目录，如下所示。

```
cd Generative-Adversarial-Networks-Projects
```

(2) 从当前目录切换到 Chapter08。

```
cd Chapter09
```

(3) 然后为本项目创建一个 Python 虚拟环境。

```
virtualenv venv
virtualenv venv -p python3 # 创建一个使用 Python 3 解释器的虚拟环境
virtualenv venv -p python2 # 创建一个使用 Python 2 解释器的虚拟环境
```

本项目会使用这个新创建的虚拟环境。每章都有独立的虚拟环境。

(4) 启用新创建的虚拟环境，如下所示。

```
source venv/bin/activate
```

启用虚拟环境之后，后续所有命令都会在其中执行。

(5) 执行如下命令，安装 requirements.txt 文件中列出的所需的库。

```
pip install -r requirements.txt
```

README.md 文件包含创建项目的更多指导。开发者经常会遇到依赖程序不匹配的问题。可以通过为每个项目创建独立的虚拟环境来解决。

这样就创建好了项目且安装了所需的依赖程序。下面处理数据集。首先介绍下载数据集及处理格式的各个步骤。

8.3 准备数据

要用的 Facades 数据集可从 http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz 下载。

该数据集包含建筑立面标注图和建筑立面真实图像。其中建筑立面一般指建筑的正立面，而建筑立面标注图是建筑立面图像的建筑学标注。下载数据集后，会对建筑立面有更多了解。执行如下命令，下载并提取数据集。

(1) 执行如下命令，下载数据集。

```
# 下载数据集之前，访问数据目录
cd data

# 下载数据集
wget
http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz
```

(2) 下载完数据集后，使用如下命令提取数据集。

```
tar -xvzf facades.tar.gz
```

数据集的文件结构如图 8-2 所示。

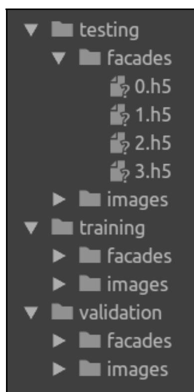


图　8-2

如上所示，数据集分为训练集、测试集和验证集。下面提取图像。

加载数据集的步骤如下。

(1) 首先创建一个包含建筑立面标注图.h5 文件的列表，以及一个包含建筑立面真实图像.h5 文件的列表，如下所示。

```
data_dir_path = os.path.join(data_dir, data_type)

# 获取包含训练图像的所有.h5 文件
facade_photos_h5 = [f for f in os.listdir(os.path.join(data_dir_path, 'images'))
                    if '.h5' in f]
facade_labels_h5 = [f for f in os.listdir(os.path.join(data_dir_path, 'facades'))
                    if '.h5' in f]
```

(2) 然后迭代（循环）这些列表，依次加载各图像。

```
final_facade_photos = None
final_facade_labels = None

for index in range(len(facade_photos_h5)):
```

以下所有代码都放在前面的 `for` 循环中。

(3) 接着加载包含图像的 h5 文件，提取实际图像的 NumPy ndarray。

```
facade_photos_path = data_dir_path + '/images/' +
                     facade_photos_h5[index]
facade_labels_path = data_dir_path + '/facades/' +
                     facade_labels_h5[index]

facade_photos = h5py.File(facade_photos_path, 'r')
facade_labels = h5py.File(facade_labels_path, 'r')
```

(4) 缩放图像至所需大小，如下所示。

```
# 缩放图像并归一化
num_photos = facade_photos['data'].shape[0]
num_labels = facade_labels['data'].shape[0]

all_facades_photos = np.array(facade_photos['data'], dtype=np.float32)
all_facades_photos = all_facades_photos.reshape((num_photos, img_width,
                                                 img_height, 1)) / 255.0

all_facades_labels = np.array(facade_labels['data'], dtype=np.float32)
all_facades_labels = all_facades_labels.reshape((num_labels, img_width,
                                                 img_height, 1)) / 255.0
```

(5) 接着将缩放后的图像添加到最终的 ndarray。

```
if final_facade_photos is not None and final_facade_labels is not
None:
final_facade_photos = np.concatenate([final_facade_photos,
                                      all_facades_photos], axis=0)
final_facade_labels = np.concatenate([final_facade_labels,
                                      all_facades_labels], axis=0)
    else:
        final_facade_photos = all_facades_photos
        final_facade_labels = all_facades_labels
```

加载图像并进行缩放的完整代码如下。

```
def load_dataset(data_dir, data_type, img_width, img_height):
    data_dir_path = os.path.join(data_dir, data_type)

    # 获取包含训练图像的所有.h5 文件
    facade_photos_h5 = [f for f in os.listdir(os.path.join(data_dir_path, 'images'))
                        if '.h5' in f]
    facade_labels_h5 = [f for f in os.listdir(os.path.join(data_dir_path, 'facades'))
                        if '.h5' in f]

    final_facade_photos = None
    final_facade_labels = None

    for index in range(len(facade_photos_h5)):
        facade_photos_path = data_dir_path + '/images/' + facade_photos_h5[index]
        facade_labels_path = data_dir_path + '/facades/' + facade_labels_h5[index]

        facade_photos = h5py.File(facade_photos_path, 'r')
        facade_labels = h5py.File(facade_labels_path, 'r')

        # 缩放图像并归一化
        num_photos = facade_photos['data'].shape[0]
        num_labels = facade_labels['data'].shape[0]

        all_facades_photos = np.array(facade_photos['data'], dtype=np.float32)
        all_facades_photos = all_facades_photos.reshape((num_photos, img_width,
                                                         img_height, 1)) / 255.0

        all_facades_labels = np.array(facade_labels['data'], dtype=np.float32)
```

```
        all_facades_labels = all_facades_labels.reshape((num_labels, img_width,
                                                         img_height, 1)) / 255.0

    if final_facade_photos is not None and final_facade_labels is not None:
        final_facade_photos = np.concatenate([final_facade_photos,
                                              all_facades_photos], axis=0)
        final_facade_labels = np.concatenate([final_facade_labels,
                                              all_facades_labels], axis=0)
    else:
        final_facade_photos = all_facades_photos
        final_facade_labels = all_facades_labels

    return final_facade_photos, final_facade_labels
```

上面的函数会加载训练集、测试集和验证集目录下的.h5 文件中的图像。

图像可视化

下面的 Python 函数可将建筑立面标注图和建筑立面图像可视化。

```
def visualize_bw_image(img):
    """
    将黑白图像可视化
    """
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.imshow(img, cmap='gray', interpolation='nearest')
    ax.axis("off")
    ax.set_title("Image")
    plt.show()
```

使用该函数对建筑立面标注图或者建筑立面照片进行可视化，如下所示。

```
visualize_bw_image(image)
visualize_bw_image(image
```

图 8-3 展示了一张建筑立面图像。



图 8-3　一张建筑立面图像

图 8-4 展示的是该图像的建筑学标注图。

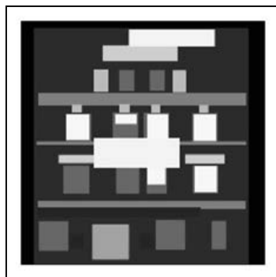


图 8-4 建筑立面图像的建筑学标注图

下面训练一个可以根据建筑立面标注图生成建筑立面图像 pix2pix 网络。首先编写生成网络和判别网络的 Keras 实现代码。

8.4 pix2pix 的 Keras 实现

前面讲过，pix2pix 由一个生成网络和一个判别网络组成。生成网络的架构受到了 U-Net 架构的启发，而判别网络的架构受到了 PatchGAN 架构的启发。下面依次实现这两个网络。

开始编写实现代码之前，首先创建一个 Python 文件 main.py，导入核心模块，如下所示。

```
import os
import time

import h5py
import keras.backend as K
import matplotlib.pyplot as plt
import numpy as np
from cv2 import imwrite
from keras import Input, Model
from keras.layers import Convolution2D, LeakyReLU, BatchNormalization, \
    UpSampling2D, Dropout, Activation, Flatten, Dense, Lambda, Reshape, concatenate
from keras.optimizers import Adam
```

8.4.1 生成网络

生成网络接收源领域 A 中维度为`(256, 256, 1)`的图像，变换成目标领域 B 中维度为`(256, 256, 1)`的图像。简单说来，它将源领域 A 中的图像变换成目标领域 B 中的图像。下面使用 Keras 框架实现生成网络。

创建生成网络的步骤如下。

(1) 首先定义生成网络所需的超参数。

```
kernel_size = 4
strides = 2
leakyrelu_alpha = 0.2
upsampling_size = 2
dropout = 0.5
output_channels = 1
input_shape = (256, 256, 1)
```

(2) 然后创建一个输入层，为网络提供输入，如下所示。

```
input_layer = Input(shape=input_shape)
```



输入层接收形状为(256, 256, 1)的输入图像，将其传递给网络中的下一层。

如前所述，生成网络由一个编码网络和一个解码网络组成。下面编写编码网络的代码。

(3) 使用 8.1.1 节列出的参数，为生成网络添加第一个卷积块。

```
# 编码网络的第 1 个卷积块
encoder1 = Convolution2D(filters=64, kernel_size=kernel_size,
                         padding='same', strides=strides)(input_layer)
encoder1 = LeakyReLU(alpha=leakyrelu_alpha)(encoder1)
```

第一个卷积块包含一个 2D 卷积层和一个激活函数。和其他 7 个卷积块不同，该卷积块没有批归一化层。

(4) 为生成网络添加余下 7 个卷积块。

```
# 编码网络的第 2 个卷积块
encoder2 = Convolution2D(filters=128, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder1)
encoder2 = BatchNormalization()(encoder2)
encoder2 = LeakyReLU(alpha=leakyrelu_alpha)(encoder2)

# 编码网络的第 3 个卷积块
encoder3 = Convolution2D(filters=256, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder2)
encoder3 = BatchNormalization()(encoder3)
encoder3 = LeakyReLU(alpha=leakyrelu_alpha)(encoder3)

# 编码网络的第 4 个卷积块
encoder4 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder3)
encoder4 = BatchNormalization()(encoder4)
encoder4 = LeakyReLU(alpha=leakyrelu_alpha)(encoder4)

# 编码网络的第 5 个卷积块
encoder5 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder4)
```

```
encoder5 = BatchNormalization()(encoder5)
encoder5 = LeakyReLU(alpha=leakyrelu_alpha)(encoder5)

# 编码网络的第 6 个卷积块
encoder6 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder5)
encoder6 = BatchNormalization()(encoder6)
encoder6 = LeakyReLU(alpha=leakyrelu_alpha)(encoder6)

# 编码网络的第 7 个卷积块
encoder7 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder6)
encoder7 = BatchNormalization()(encoder7)
encoder7 = LeakyReLU(alpha=leakyrelu_alpha)(encoder7)

# 编码网络的第 8 个卷积块
encoder8 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder7)
encoder8 = BatchNormalization()(encoder8)
encoder8 = LeakyReLU(alpha=leakyrelu_alpha)(encoder8)
```

生成网络中的编码网络部分到此为止。生成网络的第二部分是解码网络。下面编写解码网络的代码。

(5) 添加第 1 个上采样卷积块，使用 8.1.1 节列出的参数。

```
# 解码网络的第 1 个上采样卷积块
decoder1 = UpSampling2D(size=upsampling_size)(encoder8)
decoder1 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same')(decoder1)
decoder1 = BatchNormalization()(decoder1)
decoder1 = Dropout(dropout)(decoder1)
decoder1 = concatenate([decoder1, encoder7], axis=3)
decoder1 = Activation('relu')(decoder1)
```

第一个上采样块接收的输入来自编码网络的最后一层。该块包含一个 2D 上采样层、一个 2D 卷积层、一个批归一化层、一个随机失活层、一个拼接操作和一个激活函数。关于这些层的更多信息，可参考 Keras 文档。

(6) 类似地，添加后续 7 个卷积块，如下所示。


```
# 解码网络的第 2 个上采样卷积块
decoder2 = UpSampling2D(size=upsampling_size)(decoder1)
decoder2 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder2)
decoder2 = BatchNormalization()(decoder2)
decoder2 = Dropout(dropout)(decoder2)
decoder2 = concatenate([decoder2, encoder6])
decoder2 = Activation('relu')(decoder2)
```

```
# 解码网络的第 3 个上采样卷积块
decoder3 = UpSampling2D(size=upsampling_size)(decoder2)
decoder3 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder3)
decoder3 = BatchNormalization()(decoder3)
decoder3 = Dropout(dropout)(decoder3)
decoder3 = concatenate([decoder3, encoder5])
decoder3 = Activation('relu')(decoder3)

# 解码网络的第 4 个上采样卷积块
decoder4 = UpSampling2D(size=upsampling_size)(decoder3)
decoder4 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder4)
decoder4 = BatchNormalization()(decoder4)
decoder4 = concatenate([decoder4, encoder4])
decoder4 = Activation('relu')(decoder4)

# 解码网络的第 5 个上采样卷积块
decoder5 = UpSampling2D(size=upsampling_size)(decoder4)
decoder5 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder5)
decoder5 = BatchNormalization()(decoder5)
decoder5 = concatenate([decoder5, encoder3])
decoder5 = Activation('relu')(decoder5)

# 解码网络的第 6 个上采样卷积块
decoder6 = UpSampling2D(size=upsampling_size)(decoder5)
decoder6 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same')(decoder6)
decoder6 = BatchNormalization()(decoder6)
decoder6 = concatenate([decoder6, encoder2])
decoder6 = Activation('relu')(decoder6)

# 解码网络的第 7 个上采样卷积块
decoder7 = UpSampling2D(size=upsampling_size)(decoder6)
decoder7 = Convolution2D(filters=256, kernel_size=kernel_size,
                         padding='same')(decoder7)
decoder7 = BatchNormalization()(decoder7)
decoder7 = concatenate([decoder7, encoder1])
decoder7 = Activation('relu')(decoder7)

# 最后的卷积层
decoder8 = UpSampling2D(size=upsampling_size)(decoder7)
decoder8 = Convolution2D(filters=output_channels,
                         kernel_size=kernel_size, padding='same')(decoder8)
decoder8 = Activation('tanh')(decoder8)
```

最后一层的激活函数是 tanh，使生成网络生成的值限制在–1 到 1 范围内。concatenate 层用于添加跳跃连接。最后一层生成维度为`(256, 256, 1)`的张量。



concatenate 层将张量沿通道维度进行拼接。也可以另外指定一个值，沿该值对应的轴进行拼接。

(7) 最后，指定生成网络的输入和输出以创建 Keras 模型。

```
# 创建一个 Keras 模型
model = Model(inputs=[input_layer], outputs=[decoder8])
```

将生成网络的完整代码包装成 Python 函数，如下所示。

```
def build_unet_generator():
    """
    使用下面定义的超参数值创建 U-Net 生成网络
    """

    kernel_size = 4
    strides = 2
    leakyrelu_alpha = 0.2
    upsampling_size = 2
    dropout = 0.5
    output_channels = 1
    input_shape = (256, 256, 1)

    input_layer = Input(shape=input_shape)

    # 编码网络

    # 编码网络的第 1 个卷积块
    encoder1 = Convolution2D(filters=64, kernel_size=kernel_size,
                             padding='same',
                             strides=strides)(input_layer)
    encoder1 = LeakyReLU(alpha=leakyrelu_alpha)(encoder1)

    # 编码网络的第 2 个卷积块
    encoder2 = Convolution2D(filters=128, kernel_size=kernel_size,
                             padding='same',
                             strides=strides)(encoder1)
    encoder2 = BatchNormalization()(encoder2)
    encoder2 = LeakyReLU(alpha=leakyrelu_alpha)(encoder2)

    # 码网络的第 3 个卷积块
    encoder3 = Convolution2D(filters=256, kernel_size=kernel_size,
                             padding='same',
                             strides=strides)(encoder2)
    encoder3 = BatchNormalization()(encoder3)
    encoder3 = LeakyReLU(alpha=leakyrelu_alpha)(encoder3)

    # 编码网络的第 4 个卷积块
    encoder4 = Convolution2D(filters=512, kernel_size=kernel_size,
                             padding='same',
                             strides=strides)(encoder3)
    encoder4 = BatchNormalization()(encoder4)
    encoder4 = LeakyReLU(alpha=leakyrelu_alpha)(encoder4)

    # 编码网络的第 5 个卷积块
    encoder5 = Convolution2D(filters=512, kernel_size=kernel_size,
```

```
                         padding='same',
                         strides=strides)(encoder4)
encoder5 = BatchNormalization()(encoder5)
encoder5 = LeakyReLU(alpha=leakyrelu_alpha)(encoder5)

# 编码网络的第 6 个卷积块
encoder6 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder5)
encoder6 = BatchNormalization()(encoder6)
encoder6 = LeakyReLU(alpha=leakyrelu_alpha)(encoder6)

# 编码网络的第 7 个卷积块
encoder7 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder6)
encoder7 = BatchNormalization()(encoder7)
encoder7 = LeakyReLU(alpha=leakyrelu_alpha)(encoder7)

# 编码网络的第 8 个卷积块
encoder8 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same',
                         strides=strides)(encoder7)
encoder8 = BatchNormalization()(encoder8)
encoder8 = LeakyReLU(alpha=leakyrelu_alpha)(encoder8)

# 编码网络

# 解码网络的第 1 个上采样卷积块
decoder1 = UpSampling2D(size=upsampling_size)(encoder8)
decoder1 = Convolution2D(filters=512, kernel_size=kernel_size,
                         padding='same')(decoder1)
decoder1 = BatchNormalization()(decoder1)
decoder1 = Dropout(dropout)(decoder1)
decoder1 = concatenate([decoder1, encoder7], axis=3)
decoder1 = Activation('relu')(decoder1)

# 解码网络的第 2 个上采样卷积块
decoder2 = UpSampling2D(size=upsampling_size)(decoder1)
decoder2 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder2)
decoder2 = BatchNormalization()(decoder2)
decoder2 = Dropout(dropout)(decoder2)
decoder2 = concatenate([decoder2, encoder6])
decoder2 = Activation('relu')(decoder2)

# 解码网络的第 3 个上采样卷积块
decoder3 = UpSampling2D(size=upsampling_size)(decoder2)
decoder3 = Convolution2D(filters=1024, kernel_size=kernel_size,
                         padding='same')(decoder3)
decoder3 = BatchNormalization()(decoder3)
decoder3 = Dropout(dropout)(decoder3)
decoder3 = concatenate([decoder3, encoder5])
decoder3 = Activation('relu')(decoder3)
```

```
    # 解码网络的第 4 个上采样卷积块
    decoder4 = UpSampling2D(size=upsampling_size)(decoder3)
    decoder4 = Convolution2D(filters=1024, kernel_size=kernel_size,
                             padding='same')(decoder4)
    decoder4 = BatchNormalization()(decoder4)
    decoder4 = concatenate([decoder4, encoder4])
    decoder4 = Activation('relu')(decoder4)

    # 解码网络的第 5 个上采样卷积块
    decoder5 = UpSampling2D(size=upsampling_size)(decoder4)
    decoder5 = Convolution2D(filters=1024, kernel_size=kernel_size,
                             padding='same')(decoder5)
    decoder5 = BatchNormalization()(decoder5)
    decoder5 = concatenate([decoder5, encoder3])
    decoder5 = Activation('relu')(decoder5)

    # 解码网络的第 6 个上采样卷积块
    decoder6 = UpSampling2D(size=upsampling_size)(decoder5)
    decoder6 = Convolution2D(filters=512, kernel_size=kernel_size,
                             padding='same')(decoder6)
    decoder6 = BatchNormalization()(decoder6)
    decoder6 = concatenate([decoder6, encoder2])
    decoder6 = Activation('relu')(decoder6)

    # 解码网络的第 7 个上采样卷积块
    decoder7 = UpSampling2D(size=upsampling_size)(decoder6)
    decoder7 = Convolution2D(filters=256, kernel_size=kernel_size,
                             padding='same')(decoder7)
    decoder7 = BatchNormalization()(decoder7)
    decoder7 = concatenate([decoder7, encoder1])
    decoder7 = Activation('relu')(decoder7)

    # 最后的卷积层
    decoder8 = UpSampling2D(size=upsampling_size)(decoder7)
    decoder8 = Convolution2D(filters=output_channels,
                             kernel_size=kernel_size, padding='same')(decoder8)
    decoder8 = Activation('tanh')(decoder8)

    model = Model(inputs=[input_layer], outputs=[decoder8])
    return model
```

这样就创建好了生成网络的 Keras 模型。下面创建判别网络的 Keras 模型。

8.4.2 判别网络

判别网络受到了 PatchGAN 架构的启发，由 8 个卷积块、1 个全连接层和 1 个扁平化层组成。判别网络接收从维度为`(256, 256, 1)`的图像中提取的一组图像块，然后估测给定图像块的概率。下面使用 Keras 实现判别网络。

(1) 首先定义判别网络所需的超参数。

```
kernel_size = 4
strides = 2
leakyrelu_alpha = 0.2
padding = 'same'
num_filters_start = 64 # 过滤器的初始数量
num_kernels = 100
kernel_dim = 5
patchgan_output_dim = (256, 256, 1)
patchgan_patch_dim = (256, 256, 1)

# 计算图像块的数量
number_patches = int((patchgan_output_dim[0] /
                      patchgan_patch_dim[0]) * (patchgan_output_dim[1] /
                      patchgan_patch_dim[1]))
```

(2) 为网络添加一个输入层。该层接收图像块，是维度为 `patchgan_patch_dim` 的张量。

```
input_layer = Input(shape=patchgan_patch_dim)
```

(3) 然后向网络添加一个卷积层，如下所示。8.1.1 节列出了该块的配置。

```
des = Convolution2D(filters=64, kernel_size=kernel_size,
                    padding=padding, strides=strides)(input_layer)
des = LeakyReLU(alpha=leakyrelu_alpha)(des)
```

(4) 接着使用如下代码计算卷积层的数量。

```
# 计算卷积层的数量
total_conv_layers = int(np.floor(np.log(patchgan_output_dim[1]) / np.log(2)))
list_filters = [num_filters_start * min(total_conv_layers, (2 ** i))
                for i in range(total_conv_layers)]
```

(5) 然后使用 8.1.1 节列出的超参数值，再添加 7 个卷积块，如下所示。

```
# 后续 7 个卷积块
for filters in list_filters[1:]:
    des = Convolution2D(filters=filters, kernel_size=kernel_size,
                        padding=padding, strides=strides)(des)
    des = BatchNormalization()(des)
    des = LeakyReLU(alpha=leakyrelu_alpha)(des)
```

(6) 接着向网络添加一个扁平化层，如下所示。

```
flatten_layer = Flatten()(des)
```

扁平化层将一个 n 维张量变换成一个一维张量。

(7) 类似地，添加一个具有两个节点（神经元）的全连接层，使用 softmax 作为激活函数。该层接收扁平化层的张量，将其维度转换成 `(batch_size, 2)`。

```
dense_layer = Dense(units=2, activation='softmax')(flatten_layer)
```

softmax 函数将一个向量转换为一个概率分布。

(8) 接着为 PatchGAN 创建一个 Keras 模型，如下所示。

```
model_patch_gan = Model(inputs=[input_layer],
                        outputs=[dense_layer, flatten_layer])
```

PatchGAN 模型接收一个张量作为输入，输出两个张量，分别是从全连接层和扁平化层输出的。PatchGAN 已经准备就绪，但该网络无法用作判别网络，因为它只能将单个图像块分类为真或假。创建完整的判别网络的步骤如下。

(1) 首先从输入图像中提取图像块，然后逐个传递给 PatchGAN。创建一个由输入层构成的列表，其中输入层的数量和图像块数量相同。

```
# 创建一个由输入层构成的列表，其中输入层的数量和图像块数量相同
list_input_layers = [Input(shape=patchgan_patch_dim)
                     for _ in range(number_patches)]
```

(2) 接着将图像块传递给 PatchGAN，以获得概率分布。

```
# 将图像块传递给 PatchGAN，以获得概率分布
output1 = [model_patch_gan(patch)[0] for patch in list_input_layers]
output2 = [model_patch_gan(patch)[1] for patch in list_input_layers]
```

如果有多个图像块，那么 output1 和 output2 皆为由张量构成的列表。至此，应该有两个由张量构成的列表了。

(3) 如果有多个图像块，沿通道维度将它们拼接起来，以计算感知损失。

```
# 如果有多个图像块，沿通道维度将它们拼接起来，以计算感知损失
dimension to calculate perceptual loss
if len(output1) > 1:
    output1 = concatenate(output1)
else:
    output1 = output1[0]

# 如果有多个图像块，将 output2 也合并
if len(output2) > 1:
    output2 = concatenate(output2)
else:
    output2 = output2[0]
```

(4) 接着创建一个全连接层，如下所示。

```
dense_layer2 = Dense(num_kernels * kernel_dim, use_bias=False, activation=None)
```

8

(5) 然后添加一个自定义损失层，该层对接收的张量计算小批次的判别。

```
custom_loss_layer = Lambda(lambda x: K.sum(
    K.exp(-K.sum(K.abs(K.expand_dims(x, 3) -
          K.expand_dims(K.permute_dimensions(x, pattern=(1, 2, 0)), 0)), 2)), 2))
```

(6) 接着将 output2 张量传递经过 dense_layer2。

```
output2 = dense_layer2(output2)
```

(7) 然后变换 output2 的形状，使之维度为(num_kernels, kernel_dim)。

```
output2 = Reshape((num_kernels, kernel_dim))(output2)
```

(8) 接着将 output2 张量传递给 custom_loss_layer。

```
output2 = custom_loss_layer(output2)
```

(9) 然后将 output1 和 output2 拼接成新的张量，并将该张量传递经过一个全连接层。

```
output1 = concatenate([output1, output2])
final_output = Dense(2, activation="softmax")(output1)
```

使用 softmax 作为最后一个全连接层的激活函数，返回一个概率分布。

(10) 最后，为网络指定输入和输出以创建判别网络模型，如下所示。

```
discriminator = Model(inputs=list_input_layers,
                      outputs=[final_output])
```

判别网络的完整代码如下。

```
def build_patchgan_discriminator():
    """
    使用下面定义的超参数值，创建一个 PatchGAN 判别网络
    """
    kernel_size = 4
    strides = 2
    leakyrelu_alpha = 0.2
    padding = 'same'
    num_filters_start = 64 # 过滤器的初始数量
    num_kernels = 100
    kernel_dim = 5
    patchgan_output_dim = (256, 256, 1)
    patchgan_patch_dim = (256, 256, 1)
    number_patches = int(
        (patchgan_output_dim[0] / patchgan_patch_dim[0]) *
        (patchgan_output_dim[1] / patchgan_patch_dim[1]))

    input_layer = Input(shape=patchgan_patch_dim)

    des = Convolution2D(filters=64, kernel_size=kernel_size,
                        padding=padding, strides=strides)(input_layer)
    des = LeakyReLU(alpha=leakyrelu_alpha)(des)

    # 计算卷积层的数量
    total_conv_layers = int(np.floor(np.log(patchgan_output_dim[1]) / np.log(2)))
    list_filters = [num_filters_start * min(total_conv_layers,
                    (2 ** i)) for i in range(total_conv_layers)]
```

```
    # 后续 7 个卷积层
    for filters in list_filters[1:]:
        des = Convolution2D(filters=filters, kernel_size=kernel_size,
                            padding=padding, strides=strides)(des)
        des = BatchNormalization()(des)
        des = LeakyReLU(alpha=leakyrelu_alpha)(des)

    # 添加一个扁平化层
    flatten_layer = Flatten()(des)

    # 添加最终的全连接层
    dense_layer = Dense(units=2, activation='softmax')(flatten_layer)

    # 创建 PatchGAN 模型
    model_patch_gan = Model(
        inputs=[input_layer], outputs=[dense_layer, flatten_layer])

    # 创建一个由输入层构成的列表，其中输入层的数量和图像块数量相同
    list_input_layers = [
        Input(shape=patchgan_patch_dim) for _ in range(number_patches)]

    # 将图像块传递通过 PatchGAN
    output1 = [model_patch_gan(patch)[0] for patch in list_input_layers]
    output2 = [model_patch_gan(patch)[1] for patch in list_input_layers]

    # 如果有多个图像块，拼接输出以计算感知损失
perceptual loss
    if len(output1) > 1:
        output1 = concatenate(output1)
    else:
        output1 = output1[0]

    # 如果有多个图像块，将 output2 也合并
    if len(output2) > 1:
        output2 = concatenate(output2)
    else:
        output2 = output2[0]

    # 添加一个全连接层
    dense_layer2 = Dense(num_kernels * kernel_dim, use_bias=False,
                         activation=None)

    # 添加一个 lambda 层
    custom_loss_layer = Lambda(lambda x: K.sum(
        K.exp(-K.sum(K.abs(K.expand_dims(x, 3) -
              K.expand_dims(K.permute_dimensions(x, pattern=(1, 2, 0)), 0)), 2)), 2))

    # 将 output2 张量传递通过 dense_layer2
    output2 = dense_layer2(output2)

    # 变换 output2 张量的形状
    output2 = Reshape((num_kernels, kernel_dim))(output2)
```

```
    # 将 output2 张量传递通过 custom_loss_layer
    output2 = custom_loss_layer(output2)

    # 最后，将 output1 和 output2 拼接起来
    output1 = concatenate([output1, output2])
    final_output = Dense(2, activation="softmax")(output1)

    # 创建判别网络模型
    discriminator = Model(inputs=list_input_layers, outputs=[final_output])
    return discriminator
```

这样就创建好了判别网络。下面创建对抗网络。

8.4.3　对抗网络

下面创建一个由 U-Net 生成网络和 PatchGAN 判别网络组成的**对抗网络**。步骤如下。

(1) 首先初始化超参数。

```
input_image_dim = (256, 256, 1)
patch_dim = (256, 256)
```

(2) 然后创建一个输入层，向网络提供输入，如下所示。

```
input_layer = Input(shape=input_image_dim)
```

(3) 接着使用生成网络生成假图像。

```
generated_images = generator(input_layer)
```

(4) 然后从生成的图像中提取图像块。

```
# 将生成图像分割为图像块
img_height, img_width = input_img_dim[:2]
patch_height, patch_width = patch_dim

row_idx_list = [(i * patch_height, (i + 1) * patch_height)
                for i in range(int(img_height / patch_height))]
column_idx_list = [(i * patch_width, (i + 1) * patch_width)
                   for i in range(int(img_width / patch_width))]

generated_patches_list = []
for row_idx in row_idx_list:
    for column_idx in column_idx_list:
        generated_patches_list.append(Lambda(
            lambda z: z[:, column_idx[0]:column_idx[1],
                        row_idx[0]:row_idx[1], :],
                        output_shape=input_img_dim) (generated_images))
```

(5) 定判别网络，因为不需要训练它。

```
discriminator.trainable = False
```

(6) 得到了一个由图像块构成的列表。将其传递通过 PatchGAN 判别网络。

```
dis_output = discriminator(generated_patches_list)
```

(7) 最后，为网络指定输入和输出以创建 Keras 模型，如下所示。

```
model = Model(inputs=[input_layer], outputs=[generated_images, dis_output])
```

这样就创建好了一个对抗模型，由生成网络和判别网络组成。对抗模型的完整代码如下。

```
def build_adversarial_model(generator, discriminator):
    """
    创建对抗模型
    """
    input_image_dim = (256, 256, 1)
    patch_dim = (256, 256)

    # 创建输入层
    input_layer = Input(shape=input_image_dim)

    # 使用生成网络生成图像
    generated_images = generator(input_layer)

    # 从生成的图像中提取图像块
    img_height, img_width = input_img_dim[:2]
    patch_height, patch_width = patch_dim

    row_idx_list = [(i * patch_height, (i + 1) * patch_height)
                    for i in range(int(img_height / patch_height))]
    column_idx_list = [(i * patch_width, (i + 1) * patch_width)
                       for i in range(int(img_width / patch_width))]

    generated_patches_list = []
    for row_idx in row_idx_list:
        for column_idx in column_idx_list:
            generated_patches_list.append(Lambda(
                lambda z: z[:, column_idx[0]:column_idx[1],
                            row_idx[0]:row_idx[1], :],
                            output_shape=input_img_dim) (generated_images))

    discriminator.trainable = False

    # 将得到的图像块传递通过判别网络
    dis_output = discriminator(generated_patches_list)
    # 创建模型
    model = Model(inputs=[input_layer], outputs=[generated_images, dis_output])
    return model
```


至此，为生成网络、判别网络和对抗网络创建了模型，pix2pix 的训练已经准备就绪了。下面使用建筑立面数据集训练 pix2pix 网络。

8.5　训练 pix2pix 网络

和其他 GAN 类似，训练 pix2pix 网络分为两步。第一步，训练判别网络；第二步，训练对抗网络，即训练生成网络。下面开始训练。

训练 pix2pix 网络的步骤如下。

(1) 首先定义训练所需的超参数。

```
epochs = 500
num_images_per_epoch = 400
batch_size = 1
img_width = 256
img_height = 256
num_channels = 1
input_img_dim = (256, 256, 1)
patch_dim = (256, 256)

# 指定数据集目录路径
dataset_dir = "pix2pix-keras/pix2pix/data/facades_bw"
```

(2) 然后定义共用的优化器，如下所示。

```
common_optimizer = Adam(lr=1E-4, beta_1=0.9, beta_2=0.999,
                        epsilon=1e-08)
```

所有网络都使用 Adam 优化器，设置学习速率为 1e-4、`beta_1` 为 0.9、`beta_2` 为 0.999、epsilon 为 1e-08。

(3) 接着构建并编译 PatchGAN 判别网络，如下所示。

```
patchgan_discriminator = build_patchgan_discriminator()
patchgan_discriminator.compile(loss='binary_crossentropy',
                               optimizer=common_optimizer)
```

使用 `binary_crossentropy` 作为损失函数、`common_optimizer` 作为训练优化器，编译判别网络模型。

(4) 构建并编译生成网络，如下所示。

```
unet_generator = build_unet_generator()
unet_generator.compile(loss='mae', optimizer=common_optimizer)
```

使用 `mse` 作为损失函数、`common_optimizer` 作为训练优化器，编译生成网络。

(5) 接着构建并编译对抗模型，如下所示。

```
adversarial_model = build_adversarial_model(unet_generator,
                                            patchgan_discriminator)
adversarial_model.compile(loss=['mae', 'binary_crossentropy'],
                          loss_weights=[1E2, 1], optimizer=common_optimizer)
```

使用损失列表['mse','binary_crossentropy']，以及 common_optimizer 作为训练优化器，编译对抗模型。

(6) 加载训练集、验证集和测试集，如下所示。

```
training_facade_photos, training_facade_labels = load_dataset(
    data_dir=dataset_dir, data_type='training',img_width=img_width,
    img_height=img_height)

test_facade_photos, test_facade_labels = load_dataset(
    data_dir=dataset_dir, data_type='testing',img_width=img_width,
    img_height=img_height)

validation_facade_photos, validation_facade_labels = load_dataset(
    data_dir= dataset_dir, data_type='validation',img_width=img_width,
    img_height=img_height)
```

load_dataset 函数是在 8.3 节定义的。每个集合包含一组图像的 ndarray，集合的维度为(所有图像的数量, 256, 256, 1)。

(7) 添加 TensorBoard，将训练损失和网络图可视化。

```
tensorboard = TensorBoard(log_dir="logs/".format(time.time()))
tensorboard.set_model(unet_generator)
tensorboard.set_model(patchgan_discriminator)
```

(8) 接着创建一个 for 循环，其运行次数和指定训练轮数相同，如下所示。

```
for epoch in range(epochs):
    print("Epoch:{}".format(epoch))
```

(9) 创建两个列表，存储所有小批次的损失。

```
dis_losses = []
gen_losses = []
# 初始化一个变量
batch_counter = 1
```

(10) 然后在训练轮循环内部再创建一个循环,其运行次数和 num_batches 指定的次数相同,如下所示。

```
num_batches = int(training_facade_photos.shape[0] / batch_size)
for index in range(int(training_facade_photos.shape[0] / batch_size)):
print("Batch:{}".format(index))
```


训练判别网络和对抗网络的完整代码都会放在该循环中。

(11) 接着采样一小批次训练集和验证集数据，如下所示。

```
train_facades_batch = training_facade_labels[index * batch_size: (index + 1) *
                                              batch_size]
```

```
train_images_batch = training_facade_photos[index * batch_size: (index + 1) *
                                            batch_size]
val_facades_batch = validation_facade_labels[index * batch_size:
                                             (index + 1) * batch_size]
val_images_batch = validation_facade_photos[index * batch_size:
                                            (index + 1) * batch_size]
```

(12) 然后生成一批次假图像，并从其中提取图像块。使用 generate_and_extract_patches 函数，如下所示。

```
patches, labels = generate_and_extract_patches(
    train_images_batch,
    train_facades_batch, unet_generator,batch_counter, patch_dim)
```

generate_and_extract_patches 函数的定义如下。

```
def generate_and_extract_patches(images, facades, generator_model,
                                 batch_counter, patch_dim):
    # 使用真实图像和生成图像轮流训练判别网络
    if batch_counter % 2 == 0:
        # 生成假图像
        output_images = generator_model.predict(facades)

        # 创建一批次真实值标签
        labels = np.zeros((output_images.shape[0], 2), dtype=np.uint8)
        labels[:, 0] = 1

    else:
        # 接收真实图像
        output_images = images

        # 创建一批次真实值标签
        labels = np.zeros((output_images.shape[0], 2), dtype=np.uint8)
        labels[:, 1] = 1

    patches = []
    for y in range(0, output_images.shape[0], patch_dim[0]):
        for x in range(0, output_images.shape[1], patch_dim[1]):
            image_patches = output_images[:, y: y + patch_dim[0],
                                          x: x + patch_dim[1], :]
            patches.append(np.asarray(image_patches, dtype=np.float32))

    return patches, labels
```

该函数使用生成网络生成假图像，然后从生成图像中提取图像块。现在有了一个由图像块组成的列表，以及相应的真实值标签。

(13) 使用生成的图像块训练判别网络。

```
d_loss = patchgan_discriminator.train_on_batch(patches, labels)
```

提取的图像块和真实值标签用于训练判别网络。

(14) 使用如下码训练对抗模型。对抗模型会锁定判别网络，而训练生成网络。

```
labels = np.zeros((train_images_batch.shape[0], 2), dtype=np.uint8)
        labels[:, 1] = 1

        # 训练对抗模型
        g_loss = adversarial_model.train_on_batch(
            train_facades_batch, [train_images_batch, labels])
```

(15) 在每个小批次之后，增加批次计数器的次数。

```
batch_counter += 1
```

(16) 完成对每个小批次的迭代（循环）之后，将损失存储在列表 dis_losses 和 gen_losses 中。

```
dis_losses.append(d_loss)
gen_losses.append(g_loss)
```

(17) 同时，将平均损失存储到 TensorBoard，以便进行可视化，包括生成网络的平均损失和判别网络的平均损失。

```
write_log(tensorboard, 'discriminator_loss', np.mean(dis_losses),
          epoch)
write_log(tensorboard, 'generator_loss', np.mean(gen_losses),
          epoch)
```

(18) 每训练 10 轮，使用生成网络生成一组图像。

```
# 每训练 10 轮，生成图像并保存，用于可视化
if epoch % 10 == 0:
    # 采样一批次验证数据集
    val_facades_batch = validation_facade_labels[0:5]
    val_images_batch = validation_facade_photos[0:5]

    # 生成图像
    validation_generated_images = unet_generator.predict(val_facades_batch)

    # 保存图像
    save_images(val_images_batch, val_facades_batch,
                validation_generated_images, epoch, 'validation', limit=5)
```

将上面的代码块放入训练轮循环中。每训练 10 轮，会生成一批次假图像，并保存到结果目录。其中，`save_images()`是一个效用函数，定义如下。

```
def save_images(real_images, real_sketches, generated_images, num_epoch,
dataset_name, limit):
    real_sketches = real_sketches * 255.0
    real_images = real_images * 255.0
    generated_images = generated_images * 255.0

    # 只保存部分图像
    real_sketches = real_sketches[:limit]
    generated_images = generated_images[:limit]
    real_images = real_images[:limit]
```

```
# 创建一个图像栈
X = np.hstack((real_sketches, generated_images, real_images))

# 保存图像栈
imwrite('results/X_full_{}_{}.png'.format(dataset_name, num_epoch), X[0])
```

这样就在建筑立面数据集上成功训练了 pix2pix 网络。将网络训练 1000 轮，以获得质量不错的生成网络。

8.5.1 保存模型

在 Keras 中，保存模型只需一行代码，如下所示。

```
# 指定生成网络模型的路径
unet_generator.save_weights("generator.h5")
```

类似地，保存判别网络模型的代码如下。

```
# 指定判别网络模型的路径
patchgan_discriminator.save_weights("discriminator.h5")
```

8.5.2 生成图像可视化

20 轮训练后，生成网络会开始生成比较不错的图像。下面看一下这些生成的图像。

在经过 20、50、150、200 轮（从左至右）之后，图像如图 8-5 所示。

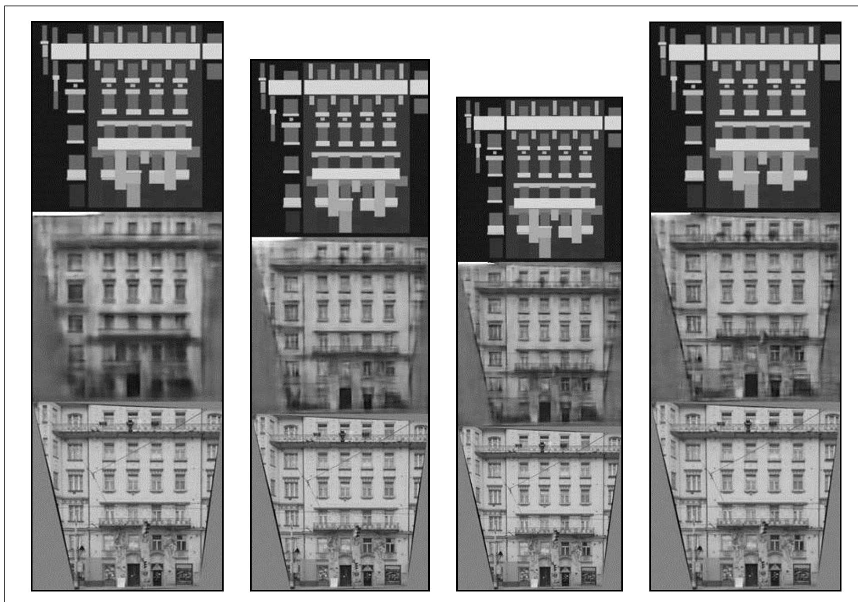


图 8-5　20、50、150、200 轮训练后生成的图像

每列从上到下分别是建筑立面标注图、生成图像和真实图像。建议将网络训练 1000 轮。一切顺利的话，1000 轮训练后，生成网络会开始生成逼真的图像。

8.5.3 损失可视化

启动 TensorBoard 服务器，将训练损失可视化，如下所示。

```
tensorboard --logdir=logs
```

然后使用浏览器访问 localhost:6006。TensorBoard 的 SCALARS 部分下包含了两种损失的曲线图，如图 8-6 所示。

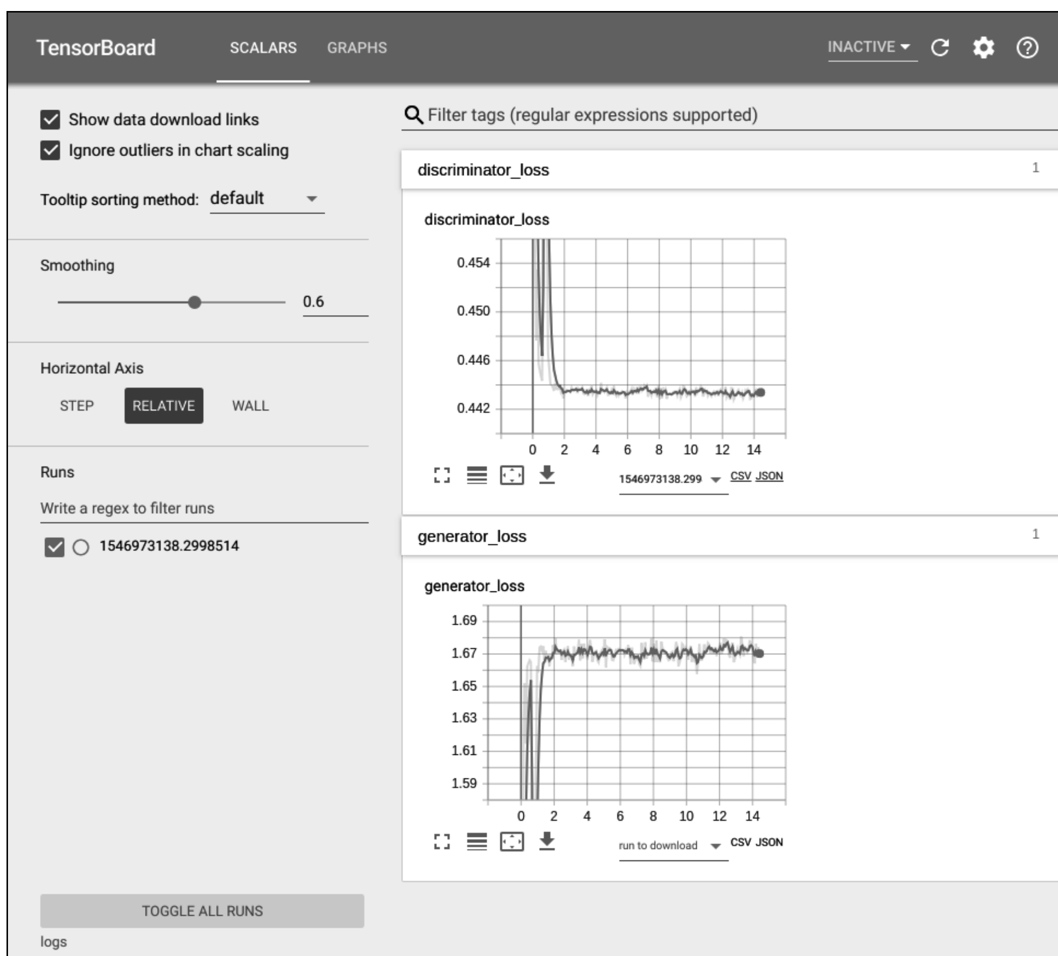


图 8-6 两种损失的曲线图

这些曲线图有助于判断是否继续进行训练。如果损失不再降低，可以停止训练，因为已经没有提升的可能了。如果损失不断提高，那么必须停止训练。尝试不同的超参数以获得更好的结果。如果损失在逐渐降低，就继续训练模型。

8.5.4　图可视化

TensorBoard 的 GRAPHS 部分包含了两个网络的图。如果这两个网络表现不佳，这些图有助于排除问题。它们还可以展示各图中的张量和不同运算的流（见图 8-7）。

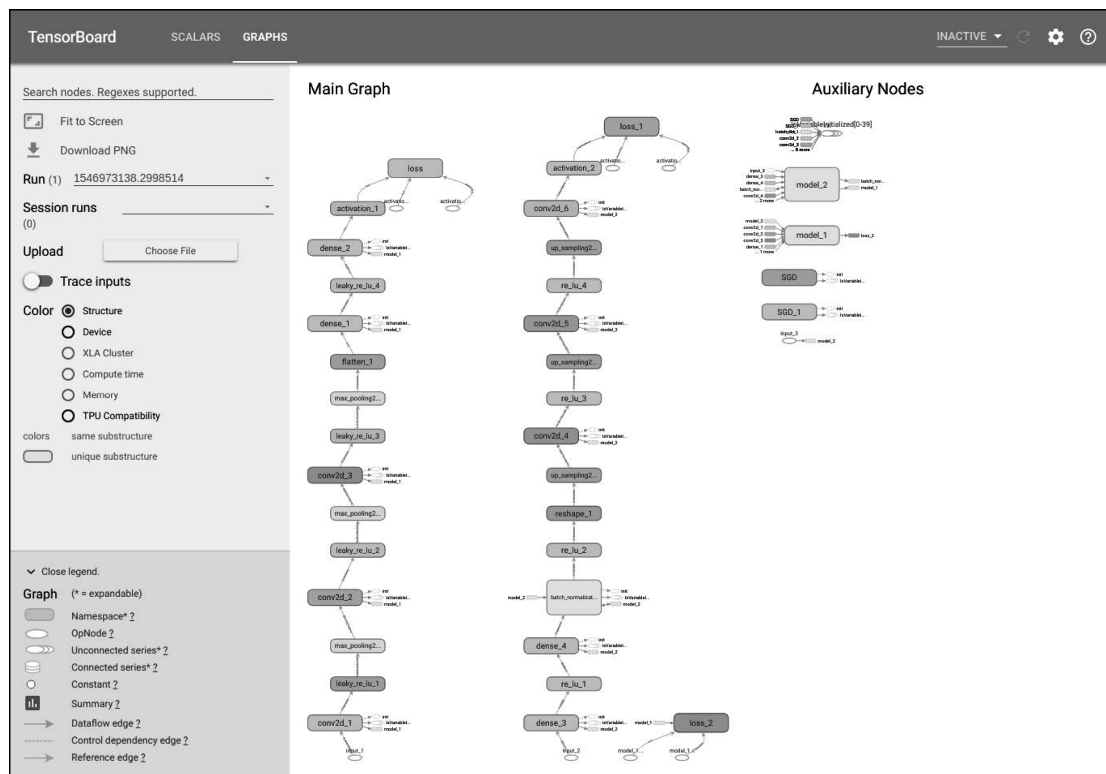


图 8-7　各图中的张量和不同运算的流

8.6　pix2pix 网络的实际应用

pix2pix 网络有很多实际应用，例如：

- 将像素级别的分段转换为真实图像；
- 将白天的照片转换为夜间的照片，或者反过来；

- 将卫星图像转换为地图图像；
- 将草图转换为照片；
- 将黑白图像转换为彩色图像，或者反过来。

图 8-8 来自官方论文，展示了 pix2pix 网络的各种应用。

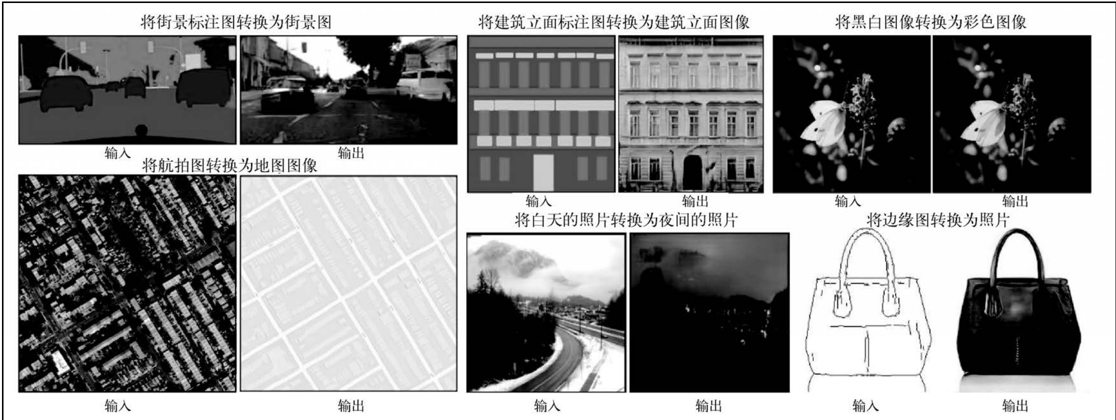


图 8-8

来源："Image-to-Image Translation with Conditional Adversarial Networks"

8.7 小结

本章介绍了 pix2pix 的概念及其架构。首先下载并准备了训练所用的数据集，然后创建了项目并用 Keras 实现了 pix2pix 网络，接着介绍了训练 pix2pix 网络所需的目标函数，然后使用建筑立面数据集训练了 pix2pix 网络，最后探讨了 pix2pix 网络的一些实际应用。

下一章会预测 GAN 的未来，探讨 GAN 领域的发展前景，以及对各个行业以及人们日常生活的影响。

第 9 章

预测 GAN 的未来

本书之前各章细致讲解了如何编写针对各种实际应用的 GAN。GAN 拥有颠覆一些行业的潜力，科学家和研究者们已经开发出了可用于构建商业应用的各种 GAN。本书探讨并实现了一些最著名的 GAN 架构。

首先回顾前面讲过的内容。

- 第 1 章简单介绍了 GAN 和相关重要概念。
- 第 2 章介绍了一种可以生成 3D 图像的 GAN——3D-GAN，并且训练了一个 3D-GAN，可以生成现实世界物体（比如飞机和桌子）的 3D 模型。
- 第 3 章介绍了可实现人脸老化的 cGAN，以及如何用它将人脸图像转换成不同年龄的图像。这一章还探讨了 Age-cGAN 的实际应用。
- 第 4 章介绍了 DCGAN，并且使用 DCGAN 生成了动画人物的面部图像。
- 第 5 章介绍了 SRGAN，并用它将低分辨率图像转换成高分辨率图像。这一章还讨论了 SRGAN 是如何解决一些非常有趣的实际问题的。
- 第 6 章介绍了 StackGAN，并用它实现了从文本到图像的合成。先是探索了一个数据集，然后用它训练了 StackGAN。这一章最后介绍了 StackGAN 的实际应用。
- 第 7 章介绍了实现图像对图像变换的 CycleGAN，并且实现了一个可以将绘画转换为照片的 CycleGAN。这一章还讨论了 CycleGAN 的实际应用。
- 第 8 章介绍了一种 cGAN——pix2pix 网络，并且训练了一个可以基于建筑标注图生成建筑立面图像的 pix2pix 网络。这一章最后介绍了 pix2pix 的实际应用。

本章讨论以下主题。

- 预测 GAN 的未来
- GAN 的潜在应用
- GAN 可以深入探索的领域

9.1　对 GAN 未来的预测

GAN 的未来具有如下特点。

- 研究者社区对 GAN 及其应用广泛接纳。
- 不错的成果：对于使用传统方法很难完成的任务，GAN 已经取得了不错的成果。比如此前将低分辨率图像转换为高分辨率图像难度很大，通常使用 CNN 来完成，而 SRGAN 和 pix2pix 等架构展现出了 GAN 在该应用上的巨大潜力。此外，StackGAN 在文本对图像合成任务上也表现良好。如今，任何人都可以创建 SRGAN，并使用图像进行训练。
- 深度学习技术的进步。
- GAN 在商业应用中的使用。
- GAN 训练过程的成熟。

9.1.1　提升现有的深度学习方法

使用有监督深度学习方法训练模型需要庞大的数据量。获取这些数据成本高昂，并且非常耗时。有时不能获得足够的数据，因为某些数据无法公开获取；有的虽然可以公开获取，但是数据集的规模又太小。这时可以诉诸 GAN。对于一个比较小的数据集，一旦在此之上训练好一个 GAN，就可以使用该 GAN 生成同一个领域的新数据了。比如处理图像分类的任务时，如果手头的数据集对于该任务来说不够大，那么可以使用已有图像训练一个 GAN，然后用它生成该领域下的新图像。尽管 GAN 目前仍然存在训练不稳定的问题，但一些研究已证明该方法可以生成逼真图像。

9.1.2　GAN 商业应用的演化

GAN 未来的商业应用将会更多。目前已经开发出了很多 GAN 的商业应用，并且取得了不错的效果。比如移动应用 Prisma 就是最早大获成功的 GAN 应用之一。在不久的将来，GAN 可能会更加大众化，届时 GAN 会提升人们日常生活的质量。

9.1.3　GAN 训练过程的成熟

从 2014 年诞生至今，GAN 一直存在训练不稳定的问题。GAN 有时完全不收敛，两个网络都从训练路径上发散。在编写本书的过程中，我曾多次遇到这个问题。研究者们为稳定 GAN 训练花费了很大精力。可以预见，随着深度学习的发展，GAN 的训练方法会日趋成熟，以后训练模型将不再会受各种问题的困扰了。

9.2　GAN 未来的潜在应用

GAN 的前景一片光明。在不久的将来，GAN 将会在以下领域得到应用。

- 基于文本创建信息图
- 设计网站
- 压缩数据
- 药物研发
- 生成文本
- 生成音乐

9.2.1　基于文本创建信息图

设计信息图是个漫长的过程，需要时间投入和专业技能。对于市场营销和社交推广而言，信息图能增色不少，信息图在社交媒体营销中起着重要作用。由于创建信息图很耗时，一些企业有时只好退而求其次，采用效果欠佳的其他策略。借助 AI 和 GAN，设计师可以更好地制作信息图。

9.2.2　设计网站

设计网站同样耗时费力，并且需要具备专业技能。GAN 可以生成初始设计启发灵感，协助设计师进行创作，从而能省下很多金钱和时间。

9.2.3　压缩数据

通过互联网可以将大量数据发送到任何地方，但这是有成本的。GAN 能提高图像和视频的分辨率，这样就可以向需要的地方发送低分辨率的图像和视频，而只占用较小的带宽，然后使用 GAN 提高数据质量。这开启了非常多的可能性。

9.2.4　研发药物

使用 GAN 研发药物可能听起来异想天开，但 GAN 已经用于生成具有特定化学和生物学性质的分子结构了。制药公司在新药研发上往往要投入了数十亿美元，而 GAN 可以大幅降低该成本。

9.2.5　使用 GAN 生成文本

GAN 在图像生成任务上已经取得了非常好的效果。GAN 的研究目前仍集中在高分辨率图像生成、文本到图像合成、风格变换、图像对图像变换等任务上。目前，使用 GAN 生成文本方向的研究不是很多，因为 GAN 的设计理念是生成连续值，所以训练 GAN 生成离散值非常具有挑战性。可以预见，未来在文本生成任务上会有更多研究。

9.2.6 使用 GAN 生成音乐

使用 GAN 生成音乐是另一个未得到足够探索的领域。音乐创作是非常具有创造性且非常复杂的过程。GAN 有潜力改变音乐产业，如果实现的话，人们可能很快就会听到由 GAN 创作的曲目了。

9.3 探索 GAN

其他 GAN 架构包括：

- BigGAN（“LARGE SCALE GAN TRAINING FOR HIGH FIDELITY NATURAL IMAGE SYNTHESIS”）
- WaveGAN（“Synthesizing Audio with Generative Adversarial Networks”）
- BEGAN（“BEGAN: Boundary Equilibrium Generative Adversarial Networks”）
- AC-GAN（“Conditional Image Synthesis With Auxiliary Classifier GANs”）
- AdaGAN（“AdaGAN: Boosting Generative Models”）
- ArtGAN（“ArtGAN: Artwork Synthesis with Conditional Categorial GANs”）
- BAGAN（“BAGAN: Data Augmentation with Balancing GAN”）
- BicycleGAN（“Toward Multimodal Image-to-Image Translation”）
- CapsGAN（“CapsGAN: Using Dynamic Routing for Generative Adversarial Networks”）
- E-GAN（“Evolutionary Generative Adversarial Networks”）
- WGAN（“Wasserstein GAN”）

不止于此，研究者们已经开发出了数以百计的 GAN 架构。

9.4 小结

本书旨在介绍 GAN 及其实际应用。人的想象力有多丰富，GAN 的潜力就有多大。如今 GAN 架构的数量非常庞大，并且日趋成熟。GAN 未来的道路仍然漫长，因为一些问题依然存在，比如训练不稳定和模式塌陷等，但也出现了各种解决方法，例如标签平滑技术、实例归一化、小批次判别，等等。希望本书有助于你实现自己的 GAN。如有任何疑问，请发送邮件到 ahikailash1@gmail.com。

版 权 声 明



微信连接



回复“机器学习”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

生成对抗网络（GAN）是一种深度神经网络架构，是当前发展迅速的机器学习领域之一，也是人工智能领域的研究热点。GAN潜力无限，能模拟任何数据分布方式，可用于构建新一代模型，其对抗学习思想也启发了深度学习领域的许多方面，并催生了一系列新技术与新应用。

本书首先介绍GAN的理论基础，然后使用Keras从头构建7个完整的GAN项目，详细讲解如何创建项目、准备数据、实现网络、训练模型以及优化模型。这7个项目分别对应一种流行的技术，难度由浅入深，讲解循序渐进。通过本书，读者能够轻松入门GAN，学会在工作或项目中构建、训练并优化实际可用的GAN模型。

- 构建3D-GAN模型，生成现实世界的3D图形
- 构建Age-cGAN模型，实现人脸验证
- 构建DCGAN模型，生成动画人物
- 构建SRGAN模型，生成高分辨率图像
- 构建StackGAN模型，基于文本描述生成逼真图像
- 构建CycleGAN模型，将绘画转换为照片
- 构建cGAN模型，实现图像对图像变换

Packt

图灵社区：iTuring.cn

热线：(010)51095183转600

分类建议 计算机/机器学习/深度学习

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-48544-1

定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks